ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA
GRADO EN INGENIERÍA INFORMÁTICA

Sistema para la gestión del reconocimiento de equivalencias entre asignaturas: aplicación web

System for managing the approval of equivalence between subjects: web application

Autor:
Adrián González Leiva
adrigonle@gmail.com
Universidad de Málaga

Tutor:
José del Campo Ávila
jcampo@lcc.uma.es
Universidad de Málaga

Co-tutor:
Mónica Trella López
trella@lcc.uma.es
Universidad de Málaga

Tutor coordinador:
José del Campo Ávila

UNIVERSIDAD DE MÁLAGA
Málaga, Diciembre 2016

Fecha de defensa:
El Secretario del Tribunal

Resumen

La movilidad estudiantil es cada vez mas frecuente entre los estudiantes. El número de acuerdos entre las distintas universidades crece acordemente, además, son varios los programas que se ofertan para ello, tanto internacionalmente (Erasmus y Única) como nacionalmente (SICUE). El alumno que decida cursar parte de sus estudios en una universidad extranjera debe realizar numerosos trámites administrativos. Uno de los más importantes es la confección del llamado acuerdo académico, que contiene las asignaturas que cursará en la universidad de destino y aquellas que les serán reconocidas en su expediente académico de la universidad de origen. Este acuerdo debe ser autorizado por las universidades de origen y destino, lo que implica que la programación docente de cada asignatura cursada en el extranjero debe ser sometida a la evaluación del profesor del área de conocimiento correspondiente de la universidad del alumno, que dará o no el visto bueno a su reconocimiento. Para la redacción del acuerdo académico el alumno contará con la ayuda de un tutor, un profesor de su centro de estudios que le ayudará a gestionar las peticiones de reconocimientos necesarias para su acuerdo académico. Actualmente, este procedimiento supone una gran carga de trabajo para todos los implicados. Este proyecto presenta la plataforma Willyfog, cuyo objetivo principal es gestionar las peticiones de reconocimiento de asignaturas, de modo que se consiga simplificar el proceso de creación del acuerdo académico coordinando el trabajo de los alumnos y del personal docente implicado (tutores, profesores de área y coordinadores de centro).

Palabras clave

Erasmus, SICUE, Única, equivalencias, aplicación web, OAuth2, OpenId Connect, MySQL, API, PHP, Java, SBT, Play framework, Slim framework, acuerdo académico

Abstract

Nowadays, student's mobility is more frequent among them. The number of agreements among different universities grow accordingly, what is more, there is a great deal of plans that are offered, both internationally (Erasmus and Unica) and nationally (SICUE). The student that decides to take part of their studies in a foreign university must accomplish many administrative processes. The main one is the learning agreement, that contains the subjects that the student will take in the destination and those that will be recognized in their academic record of the original university. This agreement must be authorized by both universities, what implies that the study plan of every subject that was studied in the foreign university must be examined by a professor of the knowledge area corresponding to the original university, whose decision will grant the recognition or not. In order to draw up the learning agreement, the student can be helped by his mentor, a teacher of their centre that will help him through this work flow. At present, this process involves a great deal of effort to all the parts that conforms the picture. This project presents the Willyfog platform, which main objective is to manage the petitions of recognition, in order to simplify the process of construction of the learning agreement, coordinating the work of the student and the staff that is involved (mentors, area professors and centre coordinators).

Keywords

Erasmus, SICUE, Unica, equivalences, web application, OAuth2, OpenId Connect, MySQL, API, PHP, Java, SBT, Play framework, Slim framework, learning agreement

Índice

Resumen **1**
Palabras clave . . . 1

Abstract **2**
Keywords . . . 2

1. Introducción **5**
1.1 Programas de movilidad estudiantil . . . 5
1.1.1 Movilidad Internacional . . . 5
1.1.2 Movilidad Nacional . . . 5
1.2 Motivación . . . 6
1.3 Procedimiento de un estudiante de movilidad . . . 6
1.4 Actores en el sistema . . . 8
1.5 Estado del arte . . . 8
1.6 Objetivos . . . 9

2. Especificación de requisitos **11**
2.1 Requisitos generales . . . 11
2.2.1 Inicio de sesión con roles . . . 11
2.2.2 Base de conocimiento de equivalencias . . . 11
2.2.3 Seguimiento de peticiones . . . 11
2.2 Requisitos de la aplicación web . . . 11
2.2.1 Registro de usuarios . . . 11
2.2.2 Creación de peticiones . . . 11
2.2.3 Sistema de comunicación . . . 12
2.2.4 Moderación de peticiones . . . 12
2.2.5 Coordinación de los centros . . . 12

3. Diseño e implementación **13**
3.1 Arquitectura . . . 13
3.1.1 Base de datos . . . 14
3.1.2 API RESTful . . . 17
3.1.3 Servidor OpenID . . . 17
3.1.4 Aplicación web . . . 17
3.1.5 Servicio externo Gravatar . . . 18
3.2 Tecnologías usadas. Comparativa . . . 18
3.2.1 Del tipado nulo al fuerte . . . 18
3.3 Dependencias del proyecto . . . 19
3.3.1 API . . . 19
3.3.2 Aplicación web . . . 20
3.3.3 Servidor OpenID . . . 21
3.4. Diseño de la API . . . 21
3.4.1 RESTful vs SOAP . . . 22

3.4.2 Endpoints . . . 22
3.4.3 Versionado . . . 23
3.5 Protección de la API . . . 23
3.5.1 Client credentials . . . 24
3.5.2 Authorization code . . . 24
3.6. Diseño de la aplicación web . . . 25
3.6.1 Arquitectura de la aplicación. MVC . . . 25
3.6.2 Integración con la API . . . 26
3.7 Integración de OAuth2 con Javascript . . . 27
3.8 Inicio de sesión OpenID . . . 28

4. Metodologías **29**
4.1. Planificación . . . 29
4.2. Pruebas . . . 29
4.3. Documentación . . . 30
4.4. Entorno de desarrollo . . . 32

5. Conclusiones **35**

Referencias **37**

Anexo **39**
A. Manual de despliegue . . . 39
I. Entorno Vagrant . . . 39
II. willyfog-api . . . 39
III. willyfog-openid . . . 41
IV. willyfog-web . . . 41
V. Dominios . . . 41
B. Manual de la API . . . 42
C. Manual de usuario de la aplicación web . . . 42
I. Usuario de nuevo ingreso . . . 42
II. Inicio de sesión . . . 44
III. Profesores de reconocimiento . . . 50
IV. Coordinadores de centro . . . 51

1. Introducción

En la actualidad, la movilidad estudiantil esta en boga. Cada vez los estudiantes tienen una consciencia más global y encuentran una muy buena oportunidad de conocer mundo en el momento en el que cursan sus estudios superiores.

Cuando un estudiante decide cursar sus estudios en el extranjero, automáticamente se le plantean multitud de ámbitos en los cuáles tendrá que organizarse (aspectos económicos, el transporte, su estancia en el extranjero, etc.). En estos aspectos, las distintas universidades (tanto nacionales como internacionales) ofrecen programas de movilidad muy diversos en los cuáles tratan de garantizar una cierta estabilidad económica mediante ayudas y acuerdos.

1.1 Programas de movilidad estudiantil

La universidad de Málaga tiene actualmente varios planes de intercambio de estudiantes disponibles. Para el interés de este trabajo solo vamos a tratar los tres principales, Erasmus, Única y SICUE, los cuales podemos dividir en dos grandes categorías: **Movilidad Internacional** [1] (Erasmus y Única) y **Movilidad nacional** [2] (SICUE). Pasamos a detallar cada uno de ellos:

1.1.1 Movilidad Internacional

En el ámbito internacional podemos elegir tanto destinos Europeos (Erasmus) como del resto del mundo (Única):

- **Erasmus**: posiblemente el plan más conocido entre los estudiantes. En realidad se trata de una familia de programas, tanto como para prácticas en el extranjero (Erasmus Prácticas), como para movilidad de personal docente. En lo que nos concierne, dentro de este programa se contemplan a todos los estudiantes que pretenden realizar sus estudios de educación superior en un país miembro de la Unión Europea cuya estancia es de entre tres y doce meses. El programa garantiza que la universidad de origen reconocerá académicamente los estudios cursados en el extranjero.
- **Única**: este programa esta diseñado para cubrir el intercambio de estudiantes con universidades no Europeas. Entre los destinos que contempla se encuentran: Norteamérica, Iberoamérica, Asia y Oceanía. Este programa contempla una duración de los estudios aproximada a los del plan Erasmus.

1.1.2 Movilidad Nacional

En cuanto a las universidades Españolas, las universidades también tienen tratados para poder garantizar el intercambio de estudiantes. El acuerdo que se encarga de esto es el programa SICUE:

- **SICUE** (Sistema de Intercambio entre Centros Universitarios Españoles): este programa cubre las necesidades de estudiantes que quieren realizar parte de sus estudios superiores en una Universidad ajena a la suya, siempre que se encuentre en el territorio nacional. Este programa, por supuesto, garantiza el reconocimiento académico en la universidad de origen. Tiene ciertas restricciones para que el estudiante pueda acogerse a este plan y contempla una duración de sus estudios en el extranjero muy semejante a la de los planes internacionales.

1.2 Motivación

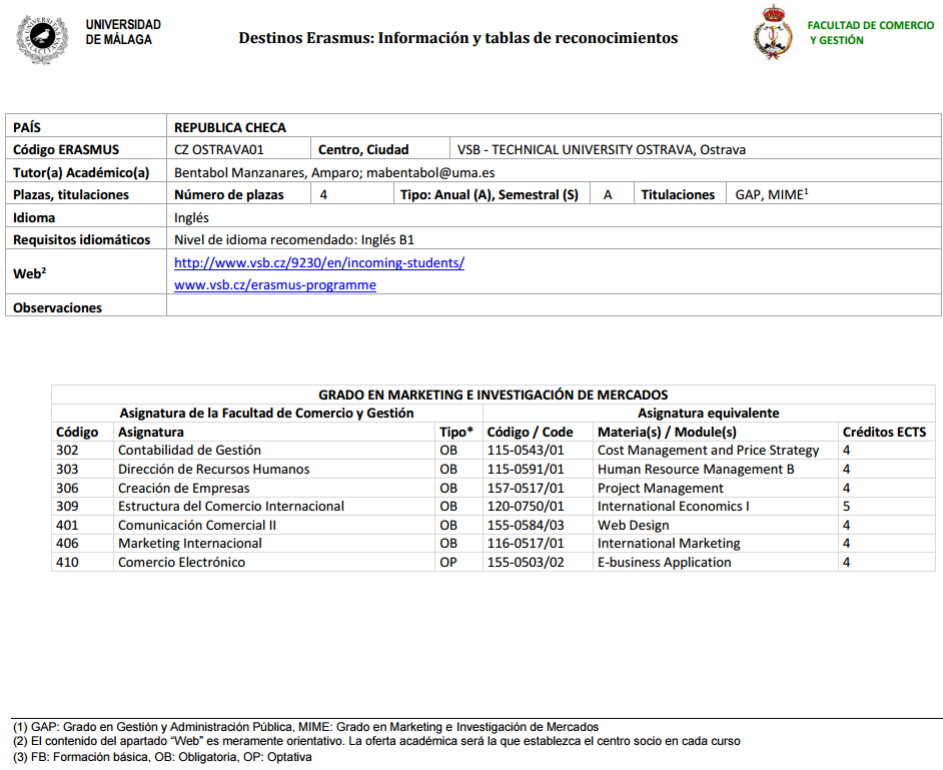
Una de las grandes preocupaciones del estudiante cuando decide cursar sus estudios en el extranjero es la de que se vean reconocidos sus esfuerzos académicos. Como hemos podido comprobar las distintas universidades tienen planes de estudios muy diversos y es debido a esta pluralidad en los destinos lo que provoca que los planes estudiantiles no sean completamente equivalentes entre ellos.

Cuando el estudiante decide cursar asignaturas en el extranjero, inicia inconscientemente un proceso de organización en el que el mismo (guiado por un tutor) deberá elaborar un plan de estudio en el extranjero, de forma que sus esfuerzos se vean reconocidos correctamente al volver a su universidad materna. Este proceso de equivalencias, aparentemente trivial debido a la existencia de acuerdos, se vuelve complejo, en gran parte debido a la generosa oferta de las distinta universidades, cuando no existen dichas equivalencias previas.

1.3 Procedimiento de un estudiante de movilidad

El primer paso que toma un estudiante cuando decide estudiar en el extranjero es comprobar en qué universidades de destino podría cursar las asignaturas que le corresponderían en su universidad de origen.

Para ello, el estudiante accede a unas **tablas de equivalencia** organizadas por universidad, donde el mismo puede observar qué asignaturas de destino equivaldrían a las asignaturas de origen, pudiendo comprobar en qué destinos cubre un mayor abanico de créditos o en qué universidades puede recibir una enseñanza más completa. En la imagen inferior se puede observar el aspecto de una tabla de equivalencias con una universidad extranjera.



UNIVERSIDAD DE MÁLAGA — Destinos Erasmus: Información y tablas de reconocimientos — FACULTAD DE COMERCIO Y GESTIÓN

PAÍS	**REPUBLICA CHECA**					
Código ERASMUS	CZ OSTRAVA01	**Centro, Ciudad**	VSB - TECHNICAL UNIVERSITY OSTRAVA, Ostrava			
Tutor(a) Académico(a)	Bentabol Manzanares, Amparo; mabentabol@uma.es					
Plazas, titulaciones	**Número de plazas**	4	**Tipo: Anual (A), Semestral (S)**	A	**Titulaciones**	GAP, MIME[1]
Idioma	Inglés					
Requisitos idiomáticos	Nivel de idioma recomendado: Inglés B1					
Web[2]	http://www.vsb.cz/9230/en/incoming-students/ www.vsb.cz/erasmus-programme					
Observaciones						

GRADO EN MARKETING E INVESTIGACIÓN DE MERCADOS

Asignatura de la Facultad de Comercio y Gestión			Asignatura equivalente		
Código	**Asignatura**	**Tipo***	**Código / Code**	**Materia(s) / Module(s)**	**Créditos ECTS**
302	Contabilidad de Gestión	OB	115-0543/01	Cost Management and Price Strategy	4
303	Dirección de Recursos Humanos	OB	115-0591/01	Human Resource Management B	4
306	Creación de Empresas	OB	157-0517/01	Project Management	4
309	Estructura del Comercio Internacional	OB	120-0750/01	International Economics I	5
401	Comunicación Comercial II	OB	155-0584/03	Web Design	4
406	Marketing Internacional	OB	116-0517/01	International Marketing	4
410	Comercio Electrónico	OP	155-0503/02	E-business Application	4

(1) GAP: Grado en Gestión y Administración Pública, MIME: Grado en Marketing e Investigación de Mercados
(2) El contenido del apartado "Web" es meramente orientativo. La oferta académica será la que establezca el centro socio en cada curso
(3) FB: Formación básica, OB: Obligatoria, OP: Optativa

6

Figura 1: Ejemplo de tabla de equivalencias de una Universidad.

En el caso de que el alumno encuentre satisfactoriamente equivalencias para sus estudios planeados, solo tendrá que rellenar un **acuerdo académico** y una vez aceptado podrá acceder a cursar sus estudios.

Si esto no ocurre, entonces el estudiante tendrá que iniciar un proceso en el que deberá recolectar información sobre el plan de estudio de la universidad en la que él cree que podrá cursar la asignatura deseada. Tendrá que iniciar con su **tutor** asignado una petición de equivalencia entre asignaturas proveyendo a este de información para que pueda valorar la petición. El tutor en este momento supervisará la petición, pudiendo filtrarla de manera que se devuelva al alumno para que añada más información o retoque algún detalle.

Si la información es correcta y suficiente, el tutor elevará la petición hacia un **coordinador de centro**, el cuál, accederá a un comité de reconocimiento para que, una vez estudiada la petición, acepte o rechace la misma, y será el coordinador el que se encargue de publicar periódicamente un tablón de notas donde el alumno podrá encontrar si su petición ha sido aceptada o no (en el caso de no aparecer en dicho anuncio).

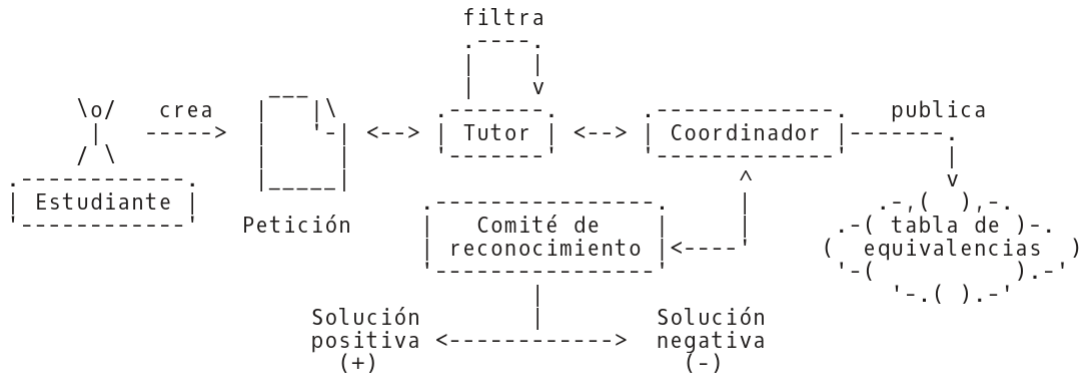


Figura 2: Diagrama del proceso.

1.4 Actores en el sistema

Como podemos observar, en el proceso hemos podido extraer tres actores principales en el sistema que procedemos a enumerar y precisar:

- El **estudiante**: previamente habiendo comprobado que no existe equivalencia posible entre la asignatura que quiere cursar y los planes de estudios foráneos, inicia una petición cuya intención es la de, una vez reconocida, establecer una equivalencia entre asignaturas.
- El **tutor**: el personal docente que se encarga de tutorizar al estudiante deberá moderar la petición (principalmente para comprobar que contiene la suficiente información necesaria para ser estudiada) y elevarla a un coordinador de centro del cuál obtendrá un *feedback*. Este hará las veces de portavoz del comité reconocedor (el cual no influye directamente en el sistema, aunque podría hacerlo) e informará al alumno del estado de su petición.
- El **coordinador**: dentro de un centro, habrá una persona que se encargará de coordinar a todos los tutores de forma que pueda garantizar el correcto funcionamiento del proceso.

Aparecen otros actores involucrados también en el proceso (como el organismo reconocedor) pero que como podremos comprobar posteriormente quedan ajenos al sistema debido a que su interacción no es directa sobre el flujo de información sino que forman parte del proceso burocrático.

1.5 Estado del arte

Actualmente, este proceso tan laborioso se realiza en su gran mayoría manualmente. Los tutores encargados de guiar a los estudiantes mantienen una comunicación constante con los alumnos por medio de correos electrónicos.

Cuando requieren información extra de los estudiantes usan como medio el **Campus Virtual** para la subida de material (como PDFs de planes de estudios de universidades). Para la organización de las distintas peticiones actualmente se usan complejas hojas de cálculo en las cuáles se establecen las distintas peticiones y el estado en el que se encuentran.

Existe una plataforma en la universidad, denominada **eVe** (Espacio Virtual Erasmus) en el cual se puede gestionar los distintos acuerdos académicos a los que los estudiantes pretenden acogerse. Por supuesto esta plataforma solo aporta valor en el momento en el que hay equivalencias tácitas entre las asignaturas de ambas universidades. En el caso de que no las haya, no se puede establecer acuerdo académico.

Por similaridad, cabe mencionar la plataforma almeriense **Ícaro** la cual sirve para gestionar la petición por parte de estudiantes para realizar prácticas en empresas. El problema de este sistema (aunque cubre un proceso similar) es que está fuertemente condicionado al proceso de las prácticas en empresas, y no ofrece flexibilidad suficiente para ser adaptado al proceso de intercambio estudiantil.

Como podemos observar, el proceso manual que actualmente se mantiene es bastante complejo y laborioso, y no existe ninguna plataforma específica que ayude a tal cometido.

1.6 Objetivos

La principal meta del proyecto *Willyfog* es la de mejorar y simplificar la comunicación de las distintas partes de dicho proceso.

Willyfog, como aplicación web sirve de punto de encuentro para los distintos actores. Tanto estudiantes, personal reconocedor de asignaturas, o coordinadores de centros, cada uno con sus propios roles y responsabilidades, pueden ver sus esfuerzos reunidos en la plataforma.

En primera instancia, los estudiantes, podrán consultar equivalencias anteriormente reconocidas en su universidad. Si no consigue satisfacer su búsqueda, podrán crear peticiones para el reconocimiento de asignaturas que tengan intención de cursar.

Por otra parte, el personal docente puede moderar este proceso, filtrando y moderando las peticiones que se creen en el sistema y guiando al estudiante. Finalmente, los coordinadores de centro podrán supervisar todo el flujo permitiendo que el mismo sea realizado con efectividad. Además, el sistema trata de dar importancia a los profesores reconocedores permitiendo un mayor filtrado de peticiones que no tienen por qué alcanzar a los coordinadores de centro.

Un punto muy interesante de la plataforma, es la de centralizar de alguna forma la base de conocimiento de equivalencias entre las distintas asignaturas de las distintas

universidades en un sistema que permite a los usuarios realizar búsquedas personalizadas cómodamente, ya que este es el punto de entrada de los estudiantes a este proceso.

2. Especificación de requisitos

Aunque ya hemos comentado brevemente los distintos requisitos que debe cumplir la plataforma, vamos a proceder en esta sección a detallarlos concienzudamente:

2.1 Requisitos generales

2.2.1 Inicio de sesión con roles

Los usuarios del sistema deben tener accesible un inicio de sesión en el sistema de forma que dependiendo del correo del usuario, se les asigne unos privilegios definidos (estudiante, profesor reconocedor o coordinador) anteriormente en el sistema. Estos privilegios definirán al usuario durante todo el uso de la aplicación pudiendo realizar una u otras actividades dentro del sistema.

2.2.2 Base de conocimiento de equivalencias

El sistema tiene que poder ofrecer, de forma pública y global, una base de conocimiento de equivalencias entre asignaturas de las distintas unviersidades que componen los distintos planes de movilidad, de forma que el usuario (estudiante) pueda comprobar los acuerdos académicos anteriormente realizados y aprobados hasta la fecha, para así poder reutilizarlos en sus estudios próximos.

El sistema también debe ofrecer la posibilidad de hacer búsquedas en esta base de conocimientos para facilitar el acceso a la información por parte de los estudiantes.

2.2.3 Seguimiento de peticiones

Los usuarios (tanto estudiantes como profesores reconocedores) deben poder realizar un seguimiento de sus peticiones abiertas en el sistema para conocer el estado en el que se encuentran. Para tal fin también contarán con un sistema de notificaciones para poder ser alertado de actualizaciones en sus peticiones.

2.2 Requisitos de la aplicación web

2.2.1 Registro de usuarios

Para que los usuarios puedan formar parte del sistema, este tiene que ofrecer un mecanismo de registro de los mismos. Este proceso tendrá que variar para los distintos roles que ocurren en el sistema, de forma que algunos registros esten controlados por usuarios con mayor privilegio. El registro de los estudiantes debe ser libre y público.

2.2.2 Creación de peticiones

En el caso de que un usuario no encuentre una equivalencia que satisfaga sus necesidades, el mismo debe de tener la posibilidad de crear peticiones de equivalencias entre

asignaturas de su universidad de origen y la universidad foránea en la que él considera que podrá cursar sus estudios.

2.2.3 Sistema de comunicación

Para facilitar la comunicación de las partes en el proceso, el sistema debe de contar con un sistema de comentarios para peticiones en el que puedan ofrecer y recibir un *feedback* del proceso de equivalencia sobre las peticiones del sistema.

2.2.4 Moderación de peticiones

El sistema debe de ofrecer la posibilidad de que los profesores reconocedores moderen las distintas peticiones del sistema de forma que puedan aceptarlas o rechazarlas según consideren oportuno.

Cuando una petición sea aprobada, el sistema tiene que añadir automáticamente la equivalencia a la base de conocimiento.

2.2.5 Coordinación de los centros

Para los coordinadores de centro, el sistema debe permitir que estos conozcan los profesores reconocedores que hay en su centro y qué asignaturas están reconociendo actualmente, así como modificar qué asignaturas son aquellas que pueden reconocer. También deben poder trazar el estado de las peticiones abiertas que ocurren en su centro.

3. Diseño e implementación

3.1 Arquitectura

Puesto que el sistema *Willyfog* esta pensado para que tenga tanto una aplicación web como móvil, la arquitectura que hemos elegido gira entorno a una **API RESTful** [3] que sirve a las dos aplicaciones de toda la lógica necesaria. Teniendo en cuenta este detalle principal, pasamos a enumerar cada una de las partes que podemos ver en el siguiente gráfico que representa la arquitectura del sistema:

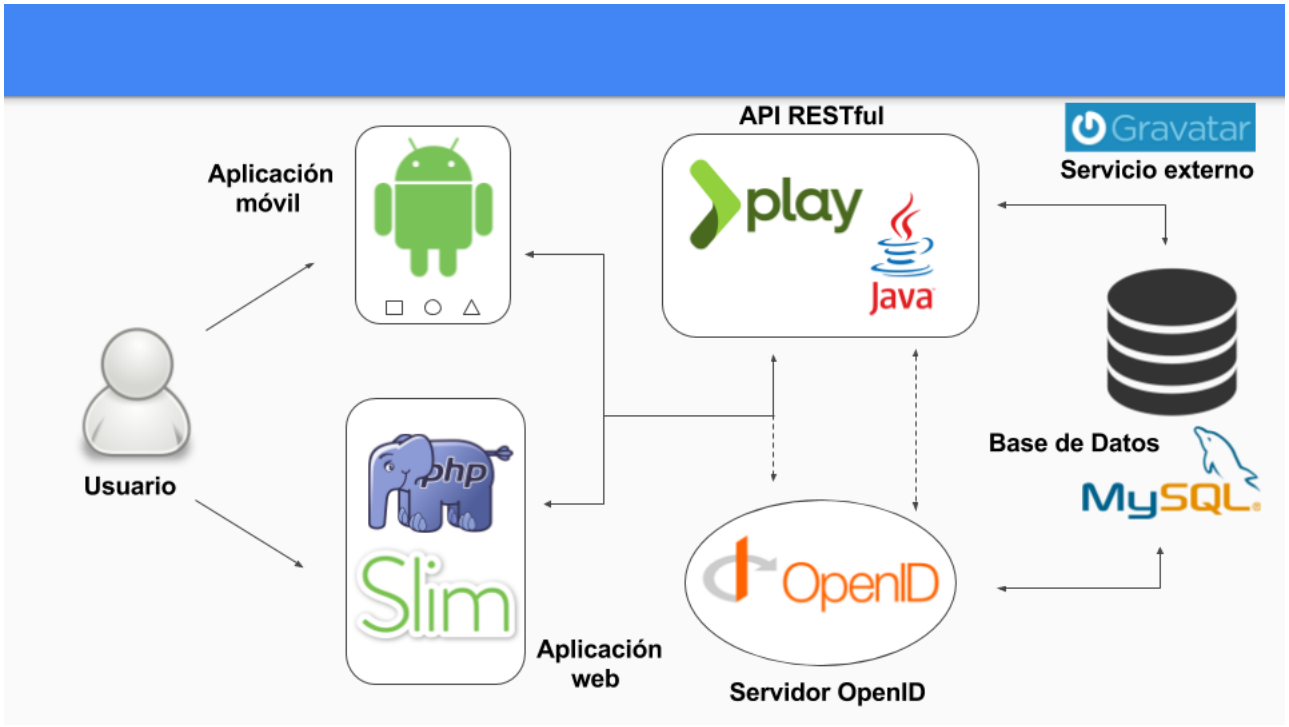


Figura 3: Arquitectura del sistema.

3.1.1 Base de datos

Puesto que nuestra aplicación debe tener una lógica centralizada, para servir de la misma forma a la aplicación móvil y web, hemos elegido una sola base de datos *MySQL*, lo cuál tiene sus puntos a favor y en contra.

Como punto positivo tenemos un menor mantenimiento, ya que ambas aplicaciones se sirven de la misma base de datos, por lo tanto no tenemos replicación de información y la persistencia esta concentrada en un solo punto.

Pero no todo es positivo, el problema de esta arquitectura es que llegado grandes niveles de tráficos, la base de datos se convertiría en un posible cuello de botella, ya que es un sistema síncrono. No se perderían datos, ya que se establecerían copias de seguridad, pero si la elasticidad del sistema.

Por sencillez, es mucho más rápido establecer una sola base de datos ya que la aplicación web y móvil comparten el mismo modelo de datos. En el caso de manejar dos fuentes de información distintas se presentaría un problema aún más difícil de resolver, como es el de la inconsistencia de los datos. Además, llegado un punto en el que la arquitectura no permita más consultas simultáneas, se podría realizar copias de la base de datos y balancear la carga entre ellas.

El modelo de datos

En cuanto al *schema*, no hemos tenido que definir relaciones demasiado complejas entre las tablas que hay en el sistema. En el modelo ER podemos observar rápidamente que las tablas más predominantes son aquellas que representan entidades básicas, como podrían ser los distintos **países** que alberga el sistema, o los distintos **grados** que ofrece una **universidad**.

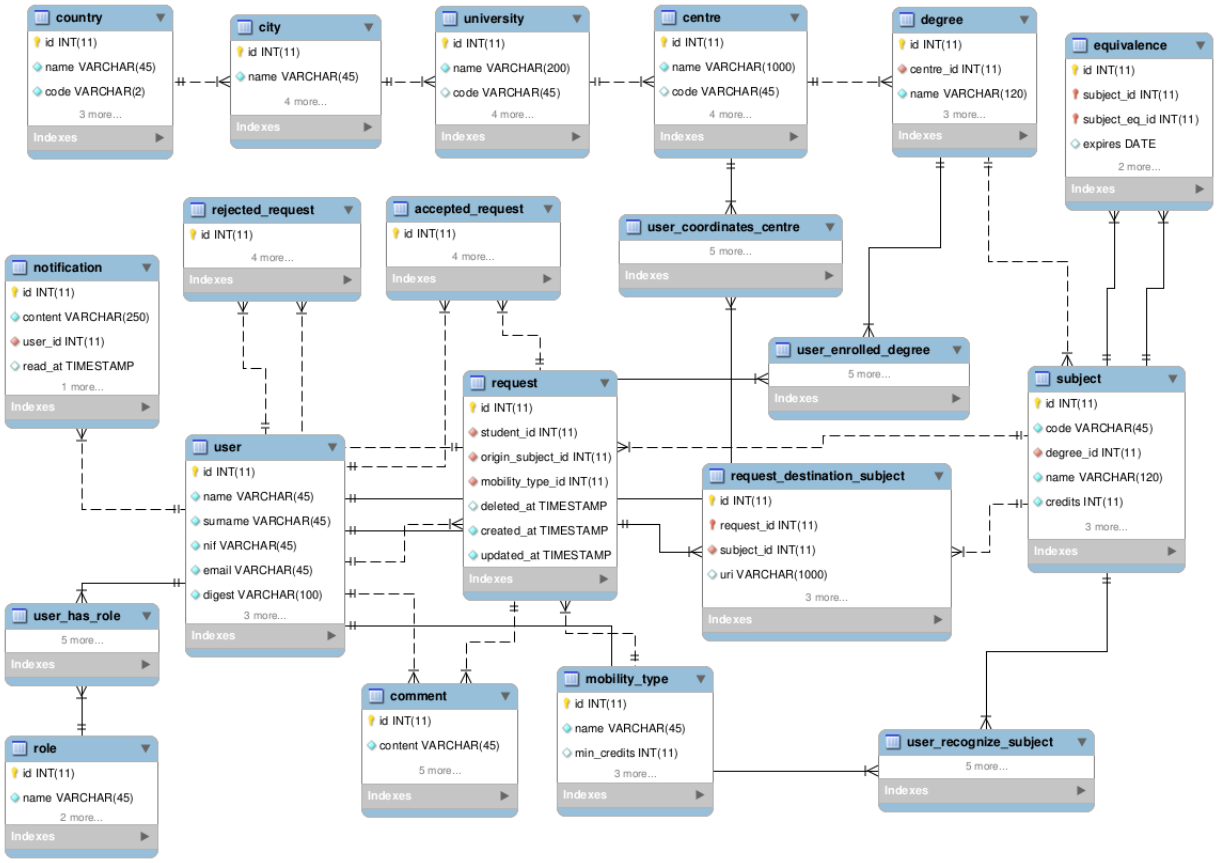


Figura 4: Diagrama ER de la base de datos (sin tablas de OAuth por simplicidad).

Uno de los detalles de diseño, que si es interesante mencionar, es la forma en la que se han modelado los distintos **estados de una petición**. Una petición se puede encontrar en tres estados distintos:

- **Aceptada** por el profesor de reconocimiento.
- **Rechazada** por el profesor de reconocimiento.
- **Pendiente**, una petición que no ha sido aceptada, ni rechazada, de hecho puede que no haya sido ni revisada.

Como primera idea, se podría pensar en representar dicho estado con una cadena de texto, o un numero que tendría un significado especial dependiendo del valor que tuviese. Este mecanismo tiene un gran problema de integridad de datos, y es que si en algun momento el sistema descuida la actualización del estado, o establece un valor inconsistente, la respuesta del mismo es incierta.

Con eso en mente, tratamos de evitar este problema de la siguiente forma. En nuestro sistema todas las peticiones están almacenadas en una tabla para tal fin. No existe ninguna columna que mantenga ninguna información sobre el estado de la misma. Para ello existen dos tablas bien diferenciadas, *accepted_request* y *rejected_request* que contienen los *ids* de aquellas peticiones que se encuentran en dicho estado. Mediante este mecanismo, ahora el estado de una petición se puede discernir de la siguiente forma:

- Una petición esta **aceptada** si su *id* se encuentra en la tabla *accepted_request*.
- Una petición esta **rechazada** si su *id* se encuentra en la tabla *rejected_request*.
- En cualquier otro caso, la petición se encontrará en estado **pendiente**.

Lo más interesante, sin duda, es que las consultas SQL que obtienen el estado del sistema se vuelven realmente sencillas y directas. De un simple *SELECT * FROM* obtendríamos todas las peticiones del sistemas. A base de *JOIN* con *accepted_request* o *rejected_request* podríamos filtrar aquellas cuyo estado viene definido por la tabla, y con una diferencia de conjuntos, podríamos obtener todas aquellas peticiones que estan pendientes. Cabe destacar que con *LEFT JOINS* y una pequeña condición, podríamos obtener el campo calculado (ya sea una cadena de texto o un valor numérico) que nos indica el estado de una petición.

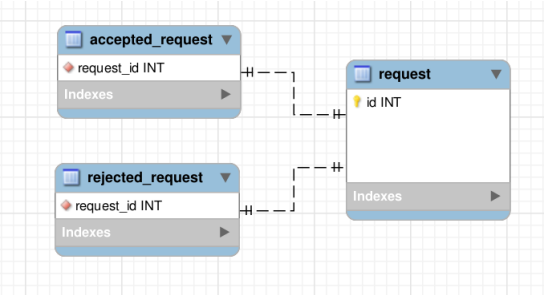


Figura 5: Diagrama ER de las tablas mencionadas.

3.1.2 API RESTful

En cuanto a la lógica de negocio, como anteriormente se ha mencionado, se tiene que concentrar en un solo punto, es por eso que se ha decidido construir conjuntamente una API RESTful en Java.

Al centralizar la base de datos, se hace natural el crear una API en torno a esta que es la encargada de leer y modificar los datos de la misma. Como valor positivo, el concentrar la lógica de negocio nos permite crear una funcionalidad una sola vez y compartirla en ambas aplicaciones. Esto vuelve la aplicación mucho más mantenible en un primer estadío.

3.1.3 Servidor OpenID

Puesto que el sistema tiene que mantener la sesión de usuarios en ambas aplicaciones, y especialmente en la aplicación web, decidimos crear un servidor OpenID [4] que se encargase del manejo de las sesiones.

Elegimos OpenID por ser el estándar actual para el manejo de sesiones en un sistema. Además nos ofrecía desde el principio un servidor OAuth2 para proteger la API (comentaremos esto más adelante). El hecho de extraer el manejo de sesiones a este servidor también nos permite concentrar cualquier tipo de login en un solo lugar, entrando en el concepto de Single Sign On [5] que tanto ha puesto de moda Google.

El servidor esta construido con PHP con la ayuda de una librería que permite implementar el protocolo con mucha facilidad.

3.1.4 Aplicación web

La parte en la que se centra este trabajo es en la aplicación web que tiene que ofrecer bastantes funcionalidades para los distintos roles que hay en el sistema.

El hecho de extraer la lógica en una API convierte la aplicación web en prácticamente un *front-end* que trata de guiar al usuario por los distintos flujos de trabajo que hay en el sistema. Es por ello que puesto que era una tarea mucho más sencilla la de *renderizar* vistas y hacer más amigable los procesos del sistema, hemos usado PHP como lenguaje

junto a *frameworks* tanto de *back-end* como de *front-end* que nos permiten crear una aplicación sencilla y fácilmente mantenible que se comunica con la API.

3.1.5 Servicio externo Gravatar

En el sistema se plantea el uso de avatares para poder identificar usuarios en el sistema. El manejo de contenido estático (como en este caso son las imágenes de un usuario) es un proceso bastante complejo.

Hay que tener en cuenta muchos detalles a la hora de permitir la subida de ficheros por parte de un usuario: limitar el tamaño de los ficheros, las dimensiones de las imágenes, incluso comprobar si lo que el usuario realmente sube tiene formato de imagen. Es por ello que decidimos usar un servicio externo denominado *Gravatar* [6], el cual asocia a nuestro correo una imagen que se comparte por todos los sitios donde nos registremos con nuestro correo personal.

Esto nos permite, en primer lugar ahorrarnos los costos tanto de servidor (para alojar las imágenes) como de complejidad en la gestión de las mismas. Por otro lado también nos permite que el usuario configure fácilmente su avatar, haciendo el uso del sistema mucho más amigable.

3.2 Tecnologías usadas. Comparativa

En esta sección vamos a pasar a comentar brevemente el por qué de las elecciones de tecnologías que hemos ido realizando en las distintas partes del sistema, centrándonos sobre todo en los lenguajes de programación.

3.2.1 Del tipado nulo al fuerte

Inicialmente elegimos PHP como lenguaje para programar la API de nuestra arquitectura. PHP tiene un típado nulo, por lo que las variables pueden referenciar a cualquier tipo de dato y este solo se conoce en tiempo de ejecución, pudiendo incluso cambiar aquí.

Este tipado en concreto es muy propenso a errores, por lo que requiere una mayor atención y testeado de las aplicaciones, y puesto que nuestra API tenía que ser robusta entendimos que no era el lenguaje más indicado.

Esto no nos hizo desechar el lenguaje, ya que había otras partes de la aplicación más triviales (en concreto la aplicación web y el servidor OpenID) donde si era un buen candidato. En cuanto a lo que respecta al servidor OpenID, realmente que la librería que usamos estuviese escrita en PHP nos condicionó fuertemente, por lo que tuvimos que usar el mismo lenguaje.

En cambio, en la aplicación web pudimos comprobar que es un candidato perfecto, ya que, como más adelante podremos observar, la naturaleza de la aplicación, al tener

que comunicarse con la API, es bastante dinámica, por lo que PHP se vuelve un buen candidato.

Una vez que nos dimos cuenta que lo que buscábamos era algun tipado en concreto, tratamos de realizar la API en Scala, ya que era un lenguaje de la JVM que estaba teniendo bastante buen recibimiento en el ambiente empresarial. Aunque es un lenguaje que todavía estamos aprendiendo y del cuál tenemos bastantes puntos a favor, tuvimos que desechar su uso rápidamente puesto que el desconocimiento relentecía en gran magnitud el ritmo del desarrollo. La aparición del lenguaje en cierta medida nos condicionó a usar el *framework* Play que permite el uso de Scala y Java de manera conjunta, y nos deja siempre una puerta abierta al salto a este lenguaje.

Por último, recordamos Java, el lenguaje que estudiamos durante la carrera, y que ofrecía un tipado fuerte. Su uso empresarial es ampliamente conocido y nuestro conocimiento inicial nos permitía avanzar bastante rápido en el desarrollo. Una de las condiciones que tuvimos es que no queríamos realizar una aplicación totalmente monolítica como podría ser una aplicación empresarial Java EE, es por ello que tratamos de movernos hacia *frameworks* más modernos y que permitiese un desarrollo más ágil.

3.3 Dependencias del proyecto

La posibilidad de usar librerías previamente creadas y testeadas por la comunidad de programadores es una gran ventaja, sobre todo a la hora de realizar un proyecto de gran complejidad. Es por ello que vamos a proceder a comentar el uso de las distintas librerías en las distintas partes de la arquitectura.

3.3.1 API

El ecosistema de librerías en Java es bastante grande y puesto que es un lenguaje ampliamente acogido en el ambiente empresarial, la calidad de estas librerías es bastante grande, es por ello que hemos usado algunos *frameworks* durante el desarrollo:

- **Play Framework** [7]: a día de hoy casi ningún programador se plantea crear una aplicación web desde cero con un lenguaje, normalmente se recurre al uso de algún *framework*. En nuestro caso optamos por Play ya que es un *framework* soportado por Lightbend [8], y que tiene un amplio abanico de casos de uso dentro de la empresa. Para nuestra API es un punto vital ya que nos provee de toda la estructura necesaria para enrutar nuestras peticiones a la API y manejarlas correctamente incluso pudiendo establecer capas de *middleware* (rutinas que se ejecutan en el momento entre que se recibe una petición y se va a manejar por la aplicación). Aunque es un *framework full-stack*, es muy modular, por lo que se puede usar a un nivel tan mínimo como es el de la construcción de una API que sirve JSON y que no *renderiza* ningún tipo de vista.
- **Guice** [9]: la inyección de dependencias (DI, *dependency injection*) es un patrón básico de diseño y que nosotros aplicamos en nuestra API. Puesto que la idea es

que el sistema maneje con facilidad un número elevado de peticiones, y que nuestro sistema sea fácilmente testeable y modular, Guice convierte la gestión de dependencias en una tarea muy sencilla debido a sus anotaciones. Cabe destacar que es una librería desarrollada por Google, cuyo uso es muy amplio en la industria y casi se esta convirtiendo en una librería *de facto*.

- **Gson** [10]: a la hora de serializar y deserializar mensajes JSON y convertirlos en objetos. Aunque Play ya incluye Jackson como librería para tal fin, encontramos que esta librería (desarrollada también por Google y con un a gran adopción entre la comunidad) era mucho más amigable a la hora de escribir serializadores personalizados para tipos de datos particulares.

- **Sql2o** [11]: para realizar consultas a la base de datos y mapear estos resultados en objetos Java. Durante el desarollo aprendimos que la base de datos se podía encargar de una gran parte de la lógica de negocio mediante consultas de SQL avanzadas. Además observamos que SQL es ya el mejor lenguaje para manejar resultados de consultas, es por ello que descartamos el uso de un ORM (*Object Relational Mapper*) como Hibernate y nos encaminamos más al uso de esta librería que simplemente encapsula el JDBC de Java para ser más sencillo de usar y que permite convertir resultados de la base de datos en objetos de Java, aunque en la mayoría de las ocasiones usamos listas de mapas que representan los resultados de las consultas y no mapeamos a ningún objeto.

- **TypeSafeConfig** [12]: el manejo de variables de entorno según estemos en desarollo o en producción es básico para un despliegue efectivo, y aunque Play nos provee de un buen instrumentaje para la gestión de las mismas, nos dimos cuenta de que cuando inyectábamos dependencias con Guice necesitábamos acceder a las variables de entorno, antes si quiera de que el *framework* entrara en juego. Es por eso que Lightbend creó esta sencilla librería y que nos permite acceder a las configuraciones en estadíos tempranos del arranque de la aplicación.

3.3.2 Aplicación web

La aplicación web tiene un cometido mucho más sencillo, y al usar tecnologías principalmente de *front-end*, necesitamos usar librerías mucho menos complejas pero sin las cuales, el desarrollo del sistema seria imposible:

- **Slim Framework** [13]: el equivalente de Play en el *front-end*. Este framework de PHP esta orientado a crear mini aplicaciones muy sencillas y rápidas. El *framework* trata de ofrecer la máxima funcionalidad con el mínimo *overhead* para poder escribir aplicaciónes web en PHP casi plano.

- **Twig** [15]: como gestor de plantillas, ya que es la solución más conocida para tal fin. Laravel ofrece un un buen gestor de plantillas como es Blade, pero que no es tan modular como podría parecer en un primer momento, ya que para usarlo hay que recurrir a más elementos del *framework*. En ese aspecto Twig, además de tener una sintaxis clara, permite su uso casi al costo de cero dependencias.

- **Guzzle** [14]: el pilar más básico de la aplicación web es la comunicación HTTP con la API. Es por eso que en este sentido hacía falta una buena libería que facilitase esta tarea, ya que Curl, la utilidad de facto en PHP es bastante engorrosa y compleja de usar. Es por ello que recurrimos a Guzzle, una librería bastante probada y que convierte la comunicación HTTP en una tarea mucho más sencilla.
- **Bootstrap** [16]: en el lado más *front-end* de la aplicación para poder maquetar. Bootstrap nos permite estructurar la web fácilmente y de forma que es muy compatible con pantallas de pequeña resolución y móviles (aunque esta no es la principal preocupación, puesto que hay una aplicación móvil).
- **jQuery** [17]: permite un manejo del DOM de HTML muy sencillo, y crear funcionalidades complejas en el lado del cliente sin tener que recurrir a elaboradas estructuras en Javascript. También facilita en gran medida el uso de llamadas AJAX, principales también en una aplicación que en última instancia se comunica con una API.

3.3.3 Servidor OpenID

En este servidor de sesiones, además de las tecnologías ya mencionadas como Slim Framework (puesto que también es una aplicación PHP) destaca el uso de una librería que nos permite la implementación del protocolo OpenID de manera muy sencilla.

La librería en cuestión se denomina bshaffer/oauth2-server-php [18]. Una de nuestras primeras ideas fue la de buscar un servidor OpenID que fuese stand-alone, es decir, que sirviese peticiones por si solo, pero al no encontrar ninguna solución que se adaptara a nuestras necesidades, recurrimos al uso de esta librería que permite la implementación rápida y sencilla del protocolo en PHP.

Con un simple *schema* de base de datos y un framework que sirva para enrutar las peticiones HTTP, se construye un servidor que vale tanto como servidor de autorización (OAuth2) como de sesiones (OpenID).

Es importante desatacar de que tanto la aplicación web como la móvil tiene una gestión de sesiones propias, y que cuando mencionamos que OpenID es un servidor de sesiones, nos referimos a que es el mismo el que decide cuando expirar las sesiones y proveer datos sobre los usuarios.

Por supuesto, como se puede intuir, es el servidor OpenID el que debe encargarse de registrar usuarios en la aplicación, para así poder aunar el punto de entrada de los usuarios en el sistema.

3.4. Diseño de la API

En cuanto al diseño de la API tuvimos pocas dudas a la hora de elegir protocolos en los que basarnos, puesto que la industria ya esta repleta de casos de éxito para protocolos como el REST. Aún así pasamos a comentar el por qué nos aferramos a esta decisión:

3.4.1 RESTful vs SOAP

En esta comparación hay que tener siempre en mente la clara diferenciación entre las épocas en los que aparecieron ambos protocolos.

Por un lado **SOAP** apareció sobre 1998. La web en aquel entonces (y no tiene por qué parecer tan evidente) era realmente distinta. En entonces la industria se movía por grandes empresas que querían crear API para sus empresas colaboradoras, con las cuáles tenía contratos muy específicos sobre qué podían consumir y que no.

Es por este carácter contractual, por el cual el protocolo es bastante inflexible. En el definimos un documento (WSDL [19]) que hace las veces de contrato, y que nos informa sobre qué operaciones podemos hacer sobre la API. Es por ello que mediante distintas utilidades (como *wsimport* [20] en Java) podemos generar código para nuestras aplicaciones de forma que realizamos llamadas RPC (*remote procedure calls*), simulando llamadas a funciones que podrían parecer que se encuentran en código local.

Lógicamente, esta visión de la web es bastante arcaica y no permite un desarrollo ágil de las tecnologías, es por eso que en la última década ha tomado gran fuerza las APIs RESTful. Las mismas se basan en el protocolo **REST** (HTTP) que define qué verbos deben de usarse para interactuar con un servidor con el cuál no se mantiene ningún tipo de sesión. Este enfoque permite una gran flexibilidad, y ya que es el protocolo subyacente en la mayoría de las comunicaciones por internet, las APIs RESTful se vuelven bastante fáciles de diseñar e intuitivas para el cliente.

3.4.2 Endpoints

Como indica el protocolo, nuestra API tiene que actuar como un servidor de distintos **recursos**. Este término tan abstracto no se refiere más que a los datos que nuestra API debe de servir.

Cada uno de estos recursos que queremos ofrecer se agruparán en **endpoints**, que no son más que URL bajo las cuáles se realizaran peticiones usando los distintos verbos HTTP (GET, POST, PUT, DELETE, etc.).

Cabe destacar que se les da un significado especial a cada uno de los verbos que se usan, por ejemplo se considera que una petición GET solo debe consultar datos, y que una petición POST crea por primera vez recursos en el servidor.

Como se puede intuir, este significado que se le da a los verbos en conjunto con el concepto de recursos, desemboca en el concepto de CRUD, en el que queremos *Create, Read, Update and Delete* para distintos modelos de datos. En nuestro sistema los recursos son nuestros distintos modelos (por ejemplo, universidades en el sistema, o usuarios) y asociamos cada uno de los verbos a una de las acciones anteriormente descritas.

Lógicamente todo esto en una simple asociación, y no hay ninguna especificación o documento que te obligue a tomar estas pautas, es por ello que en nuestra API (como en otras muchas de empresas conocidas) a veces esta correspondencia no es exacta, y un método POST realmente no tiene por qué crear ningún recurso, pero si nos provee por ejemplo de otras ventajas como la de poder incluir un cuerpo en el mensaje HTTP, y que la petición no quede registrada en el historial de búsquedas.

Por todo lo anteriormente mencionado, nuestros *endpoints* se corresponden en gran medida con los distintos modelos de datos que hay en nuestro sistema. Ello se puede comprobar realmente bien en el Anexo B donde se documentan cada uno de ellos, y los métodos que hay que usar.

3.4.3 Versionado

Una de las ventajas (o desventajas) que ofrecía SOAP es la de establecer un contrato tácito entre los implementadores sobre que operaciones se pueden realizar. Es por ello que en el caso de que haya que actualizar el servicio, simplemente definiendo un nuevo WSDL donde aparecen nuevas operaciones sobre la API se puede notificar a todos los implementadores sobre nuevas funcionalidades, y ninguna de las implementaciones anteriores deja de funcionar.

Este hecho no es tan directo en REST, y es por ello que se recurre al versionado de las APIs. Este concepto es muy sencillo, ya que todos los endpoints (y en definitiva esta es la unidad básica de la API) son urls, se puede incluir algun tipo de cadena de texto que identifique la versión de la API sobre la que se está trabajando.

Por ejemplo, un endpoint podría ser */users/new* pero si se detecta que se puede realizar alguna mejora en el mismo, sería imposible cambiar la lógica del endpoint sin romper las implementaciones anteriores.

Es por ello que se incluye en la dirección una versión de la API del estilo de */v1/users/new* de forma que si es necesario cambiar el endpoint, estos cambios serían públicos bajo la ruta */v2/users/new* (por supuesto manteniendo la lógica anterior) de forma que todas las implementaciones anteriores se mantienen intactas.

3.5 Protección de la API

Una de las partes más importantes de exponder una API al público es la de proteger qué usuarios pueden acceder a sus servicios. En este ámbito OAuth2 se ha vuelto un protocolo estándar para tal fin. Hoy en día la mayoría de APIs públicas se basan en este protocolo puesto que ofrece una gran flexibilidad para restringir el uso de la misma, y es por ello que hemos decidido usarlo.

Vamos a diferenciar entre tres actores básicos:

A El usuario que trata de acceder a la API

B El servidor de OAuth2

C La API que exponemos al público

Cuando A trata de acceder a C, comprobará que la misma tiene como requisito que en la petición se incluya un access_token. Este término nos sirve para denominar una cadena de carácteres aleatoria (y expirable) que tiene información útil para saber si la petición que se realiza es correcta y debemos atenderla o no es confiable. Para que el usuario pueda obtener un access_token tendrá que realizar una petición a B ofreciendo algún tipo de información que variará según el flujo de autorización.

OAuth2 ofrece varios flujos de autorización pero no todos eran de utilidad para nuestro sistema, es por ello que pasamos a comentar los básicos. Para más detalle sobre cómo se construye la comunicación con OAuth2 entre los implementadores y la API, en la sección 5.2 lo comentaremos con más detalle.

3.5.1 Client credentials

Este es el flujo de OAuth2 más sencillo e inmediato disponible, pero a pesar de eso permite una gran flexibilidad.

1. En este caso, a A se le asigna un Client ID que no es más que un identificador publico (del estilo de una clave pública en criptografía asimétrica) y un Client Secret que se podría considerar como la contraseña del usuario. Ambos campos son cadenas de carácteres aleatorios, de una longitud variable, que nos permite identificar a una aplicación únicamente en nuestro servidor.

2. Para obtener un access_token proveemos a B de ambas cadenas, a modo de usuario y contraseña, de forma que B puede reconocernos como un cliente de C y asociarnos un access_token con el cuál poder realizar peticiones autorizadas a C.

3.5.2 Authorization code

Este flujo es un poco más complejo, y es justamente la mejora que ofrece OAuth2 sobre OAuth (el protocolo anterior). En este caso tenemos que distinguir un usuario más en nuestro esquema que es D. Usuario el cual mantiene sus recursos en la API.

Conocer el flujo de datos anterior nos permite conocer este mejor, ya que nuevamente se le asocia un Client ID y un Client Secret a A. En este caso concreto, A intentará acceder a un recurso en B cuyo dueño (D) no tiene por qué haber autorizado el acceso. Es por ello que en este flujo necesitamos el consentimiento tácito de D para acceder a sus datos. Esto se realiza mediante una redirección en el flujo anterior de datos.

1. A redirecciona a una url de B (se denomina authorization url) con los datos Client ID y Client Secret, a qué dirección se debe realizar el callback una vez que el usuario D nos dé su respuesta y algunos parámetros extra para mejorar la seguridad que no son de importancia para entender el procedimiento.

2. Al usuario D se le muestra un formulario en el que explícitamente indica que autoriza (o no) el acceso a sus datos a A.

3. Una vez que C indica una respuesta, B realiza una redirección al callback el cuál nos indico A al principio del flujo, con un código como parámetro (denominado authorization code) el cual indica el estado afirmativo de autorización, o un mensaje de error en caso contrario.

4. Ejecuta el callback, y al recibir el código como parámetro, usa este para llamar a otro endpoint de B donde obtiene un access_token si la autorización ha sido positiva.

Este flujo puede resultar familiar para el proceso de login, y es que justamente OpenID se basa en el mismo (pero ofreciendo cierta información extra, ya que asocia un access_token a un usuario en concreto) para poder implementar el inicio de sesión.

3.6. Diseño de la aplicación web

Como mencionamos anteriormente, la aplicación web hace uso de un framework, Slim, que ya esta diseñado siguiendo el patrón MVC (Modelo Vista Controlador). Éste es el patrón que usan hoy en día la mayoría de los frameworks actuales, ya que se ha comprobado que se ajusta realmente bien al diseño de aplicaciones web, es por ello que pasamos a comentarlo más en detalle en las siguientes secciones.

3.6.1 Arquitectura de la aplicación. MVC

El patrón MVC divide la aplicación en tres partes principales y bien diferenciadas:

- **Modelo**: es aquella parte de la aplicación que se encarga de comunicarse con la capa de persistencia (o modelo de datos) y conseguir los datos necesarios para que la aplicación pueda servir información. Normalmente (ya que la mayoría de lenguajes son orientados a objetos) esta parte esta constituidas por clases que tratan de equivaler el modelo de datos, ya sea tablas en la base de datos, o cualquier otro tipo de persistencia que tenga la aplicación. Es de este concepto de donde nacen la mayoría de los ORM, que tratan de mapear el *schema* de la base de datos en objetos de un lenguaje de programación. Típicamente es aquí donde se encontrarán todas las consultas a la base de datos, en el caso de que la haya.

- **Vista**: esta constituido por toda la capa de visualización de la aplicación. Típicamente esta parte de la aplicación estará comprendida entre todas las vistas HTML (o escritas en un sistema gestor de plantillas como puede ser Twig, pero que al final se traducen a HTML) y que permite mostrar los datos dinámicos que obtiene el modelo al usuario.

- **Controlador**: hemos hablado de la parte que se encarga de conseguir la información y de la que se encarga de mostrarla, por ultimo nos faltaba una sección que se encargase de conectar ambas partes. Esa es la principal función del controlador, debe conocer los dos límites de la aplicación y comunicarlos de forma que diriga el flujo de datos. Típicamente el controlador recibirá una petición, llamara al modelo para obtener los datos necesarios, hará algun tipo de lógica necesaria, y facilitará los datos a la vista para que los muestre al usuario.

Una vez establecida la jerarquía de nuestra aplicación, es importante conocer que en nuestor caso no teníamos, por ejemplo, base de datos (puesto que está detras de la API) con la que comunicarnos directamente, por lo que en nuestro caso, nuestro modelo (que sigue cumpliendo la misma función) varía ligeramente de la media. En nuestro caso no había consultas a la base de datos, y cambiamos el *driver* encargado de comunicarse con la base de datos con una librería, como es Guzzle, que se encarga de realizar peticiones HTTP, puesto que nuestra comunicación era puramente a través de internet.

También cambiamos las tablas de base de datos a las que consultar, por *endpoints* de la API a los que pedir información, y los *JOIN* de SQL por consultas a más de un endpoint. Todas estas diferencias nos hacen entender que una de las principales partes de la aplicación web era que la comunicación con la API fuese efectiva, pues esta se convertía en nuestro modelo, y además, lo hacía de forma dinámica, de manera que no necesitábamos conocer la base de datos subyacente, sino que podíamos construir nuestor propio modelo a partir de las especificaciones de la API, o como realizamos, confiar plenamente en el "schema"que nos proveía la API.

Esta es otra de las razones por las cuales PHP, como lenguaje dinámico se ajustaba realmente bien a esta parte del sistema, puesto que era capaz de manejar las posibles situaciones de comunicación o fallo en una petición de manera bastante cómoda, gracias a su carácter dinámico, y también ser bastante flexible a cambios en el "schema".

Nótese que cuando hablamos de "schema"no nos referimos al modelo que puede existir en una base de datos, sino al modelo intrínseco que tiene la API por la estructura que tienen en la respuesta los distintos recursos que nos ofrece.

3.6.2 Integración con la API

El carácter "dinámico"de nuestra aplicación web, puesto que no se comunica con ninguna capa de persistencia sino con una API a través de internet, convierte la integración con la misma en una parte vital de la aplicación.

Básicamente nos encontramos con dos flujos principales (los cuáles lógicamente van a ser semejantes a los de OAuth2 que detallamos en secciones anteriores): cuando la aplicación web trata de obtener recursos de la API sin tener todavía ninguna información de usuario (client credentials) y cuando una vez iniciado sesión un usuario, la aplicación web se comunica con los recursos de la API (authorization code) de un usuario en concreto.

Desde la perspectiva de la aplicación web, no exponemos la API a otros consumidores, ni realizamos llamadas desde el código Javascript (el cuál es tremendamente inseguro, puesto que esta en el lado cliente) por lo que realmente solo con el flujo de client credentials nos hubiese sido suficiente, puesto que la aplicación web y la API establecen una comunicación servidor a servidor segura, por lo tanto el uso de los dos flujos anteriormente mencionados se vuelven equivalentes en algunas partes de las API.

Con la entrada en juego de la aplicación móvil, y la implementación de OpenID, entra en acción el flujo de *authorization code*, por lo que es bastante clarificador y se convierte en una buena prueba de concepto.

3.7 Integración de OAuth2 con Javascript

Anteriormente hemos mencionado que el código Javascript que hace peticiones a la API es bastante inseguro, y es por ello que merece una mención especial en esta sección del documento. Pero primeramente, vamos a demostrár el por qué de esta inseguridad.

Desde el punto de vista de la autorización, Javascript es una pesadilla (y con Android ocurre de manera similar, puesto que el código se puede decompilar), ya que el código fuente se encuentra, y se ejecuta, del lado cliente y puede ser modificado con gran facilidad. Este hecho convierte cualquier mecanismo de los anteriormente mencionados en un despropósito, puesto que en cualquiera de los casos, necesitaríamos mantener en el código, a la vista pública, un Client Secret el cual se debería tratar como una contraseña. Este hecho permitiría que cualquiera pudiera acceder a estos parámetros y realizar llamadas autorizadas a la API, conviertiéndose en un impostor de nuestra aplicación web.

La especificación de OAuth2 propone un flujo alternativo para estos casos, que como se puede intuir del escenario que presentamos, no deja de ser bastante arriesgado, y es el Implicit Flow. En el mismo, se le asigna a la aplicación Javascript consumidora un Client ID que si es un identificador único y público y se puede mantener en el código, y se le provee de un *endpoint* donde puede pedir access_token proveyendo simplemente de su Client ID.

La seguridad de este protocolo reside en que los access_token que sirven a estas aplicaciones "inseguras" tienen un tiempo de vida muy corto, por lo que si una aplicación malintencionadamente accede a los recursos, solo podrá hacerlo por un tiempo muy limitado. Además, típicamente (y esto no queda tan claro en la especificación) estas aplicaciones tendrán un acceso muy restringido a los recursos de la API. De hecho, las mismas solo deberían tener acceso a los endpoint de consulta de datos, puesto que no queremos que una aplicación no confiable cree recursos en nuestro sistema, o mucho peor, sea capaz de borrarlos.

Por otro lado, el manejo de access_token en el código Javascript se vuelve bastante engorroso, puesto que el lenguaje no permite mecanismos cómodos de cacheado (por

ejemplo para guardar dicho token) por lo que en nuestra aplicación web le dimos otro enfoque distinto al Implicit Flow.

En nuestro caso nostros hicimos que el código Javascript llamase a rutas de nuestra aplicación web, como si de su propia API se tratase, y que esta a su vez si que se comunicaba con la API en última instancia. Esto nos permitía dos ventajas claras: por una parte, teníamos cubierta la autorización con la API, puesto que la aplicación web ya es capaz de establecer una comunicación segura, y por otro lado, nos permite ocultar los endpoints de la API a posibles curiosos que quisieran realizar ingeniería inversa desde el código Javascript.

3.8 Inicio de sesión OpenID

Por último, nos queda por mencionar OpenID, el cuál debido a que en nuestro sistema convergían tanto aplicación web, como móvil, y que desde ambas se debía de permitir el inicio de sesión, se tornaba un requisito indispensable.

El flujo de OpenID para inciar sesión es prácticamente equivalente a Authorization Code Flow de OAuth2 (puesto que se basa en el mismo) pero con unas pequeñas salvedades que pasamos a comentar:

- En el caso de Authorization Code lo que se mostraba al usuario era un formulario para dar autorización a la aplicación a acceder a sus datos. En este caso, puesto que solo tenemos un sistema (aunque OpenID permite aunar incios de sesiones de varias aplicaciones) al loguear correctamente al usuario, consideramos esta como la autorización necesaria.
- OAuth2 provee de un access_token cuando se llama al *endpoint* correspondiente con el authorization code. En este caso, el servidor OpenID responde con un JWT (JSON Web Token) con mucha más información, puesto que ademas relaciona el access_token con un usuario de nuestro sistema.

4. Metodologías

4.1. Planificación

En cuanto a la planificación del proyecto, hemos puesto en práctica una de las metodologías ágiles que hoy está más en boga, como es **SCRUM**. Por ser un equipo realmente reducido (solo dos integrantes) y trabajando remotamente, divergimos ligeramente sobre la metodología original, por eso procedemos a describir los detalles:

- Elegimos **dos semanas** como duración de nuestros **sprints** por varias razones:
  - Era una duración media que nos permitía establecer un ritmo de **desarrollo ligero**, y que permitía trazar cada sprint con más **detalle**, ya que no podíamos establecer demasiadas tareas en tan poco tiempo.
  - Con esa duración, en un año hay aproximadamente 26 *sprints*, tantos, como letras en el abecedario inglés, lo cual nos permitía asignar a cada *sprint* una letra, y así poder ubicarlo fácilmente dentro del año.
- No realizábamos *stand-ups* diarias debido al carácter remoto del desarrollo, pero si que tratábamos de mantener reuniones casi diarias a través de comunicación VoIP.
- No había ningún *Scrum Master* puesto que solo éramos dos integrantes en el equipo y preferíamos mantener una jerarquía horizontal. Esto nos permitió a ambos poder tomar el rol indirectamenre, y así aprender mejor la metodología

Para ayudarnos con esta metodología usamos JIRA como gestor de tareas, de forma que teníamos siempre una *pizarra* disponible para trazar las tareas, y que nos permitía generar gráficos tras un sprint, el *burndown chart* que nos indicaba cómo había ido el *sprint*, si nos habíamos cargado demasiado de tareas, si habíamos podido cumplirlas o si necesitábamos realizar más funcionalidades en dicho periodo de tiempo.

4.2. Pruebas

Inicialmente, nuestra intención era la de seguir una metodología **TDD** (*Test Driven Development*). Rápidamente descubrimos que nuestro desconocimiento en el capo del testeo relentecía de sobremanera nuestro desarrollo, y en vez de crear un producto mucho más seguro, generaba un sistema que no estaba testeado completamente, y el cuál tardaría mucho más en ser desarrollado.

Es por eso que adoptamos una posición mucho más conservadora, y dejamos de seguir esta metodología. En su lugar, y a modo de alternativa, desarrollábamos nuestras funcionalidades sin *tests* y ya que teníamos un registro de las funcionalidades que desarrollábamos en nuestros *sprints*, dedicábamos los dos últimos días de cada *sprint* en testear las funcionalidades que desarrollábamos.

Esto nos permitía seguir un desarrollo mucho más ágil, sin dejar a un lado la estabilidad del sistema, que indudablemente sería mucho más alta en el caso de haber realizado tests. Pese a todo esto, el sistema ha sido desarrollado con la testeabilidad siempre en mente, mediante distintas técnicas como Inyección de Dependencias, etc. y al ser un proyecto *Open Source* esta abierto a que se escriba una *suite* de tests que ponga a prueba su fiabilidad.

4.3. Documentación

Un aspecto muy importante de la API, sobre todo desde el punto de vista de un implementador, es la de tener a mano una documentación de que rutas hay disponibles y que parámetros reciben, puesto que navegar por el código de la API para discernir estos aspectos es una tarea bastante compleja y que requeriría mucho tiempo.

Hay varias utilidades que nos ayudan a escribir documentación sobre API RESTful y que generan luego documentos en varios formatos que pueden ser fácilmente visualizados, pero nos quedamos con dos candidatos principales como son RAML [21] y Swagger [22], los cuáles vamos a pasar a comentar y explicar por cuál nos decidimos:

Primeramente probamos **RAML** el cuál ofrece una forma realmente cómoda de documentar la API, que es mediante un documento YAML el cual tiene un formato especifico y que se centra en aspectos bien definidos de la API, como por ejemplo el mecanismo que tenía de autorización, los distintos endpoint que ofrece, o qué tipo de respuestas sirve cada uno.

Como punto fuerte, RAML permitía una documentación muy ligera pues tiene muy buenos mecanismos de reutilización y organización del documentos, como la definición de schemas JSON genéricos que podían reutilizarse a lo largo del fichero, o ejemplo de las respuestas que ofrecía la API. Como caso de uso destacar que la documentación de la API pública de Spotify esta especificado con esta utilidad.

Mas tarde, investigamos sobre **Swagger** el cuál conocimos anteriormente y que ya se había mostrado un buen candidato para ocasiones anteriores, que se presentaba como el candidato único para documentar APIs, pero cuyo desventaja principal era que sus mecanismos de reutilización no eran tán prácticos. En este segunda visita a la especificación, la cuál habían renovado en Agosto de este año, descubrimos gratamente que de un compendio de empresas había nacido una iniciativa denominada *Open API Initiative* [23] que trata de buscar un lenguaje genérico para especificar API, y el candidato como no podía ser de otra forma era Swagger.

Es por ello, y por que los mecanismos de documentación, menos pequeñas diferencias, son realmente semejantes entre ambas utilidades (ambos documentos se pueden escribir en YAML) por lo que elegimos finalmente Swagger que ofrecía una gran potencia expresiva y que apunta a convertise en el estándar para documentar APIs RESTful. Una herramienta

que destacamos para la visualización es Swagger UI [24], la cual permite tanto navegar por la documentación como probar los distintos endpoints directamente desde una interfaz web. RAML también tiene la *API Console* pero, es en este caso particular, que la de Swagger gana la batalla al estar más elaborada. La documentación final se puede consultar en el Anexo B.

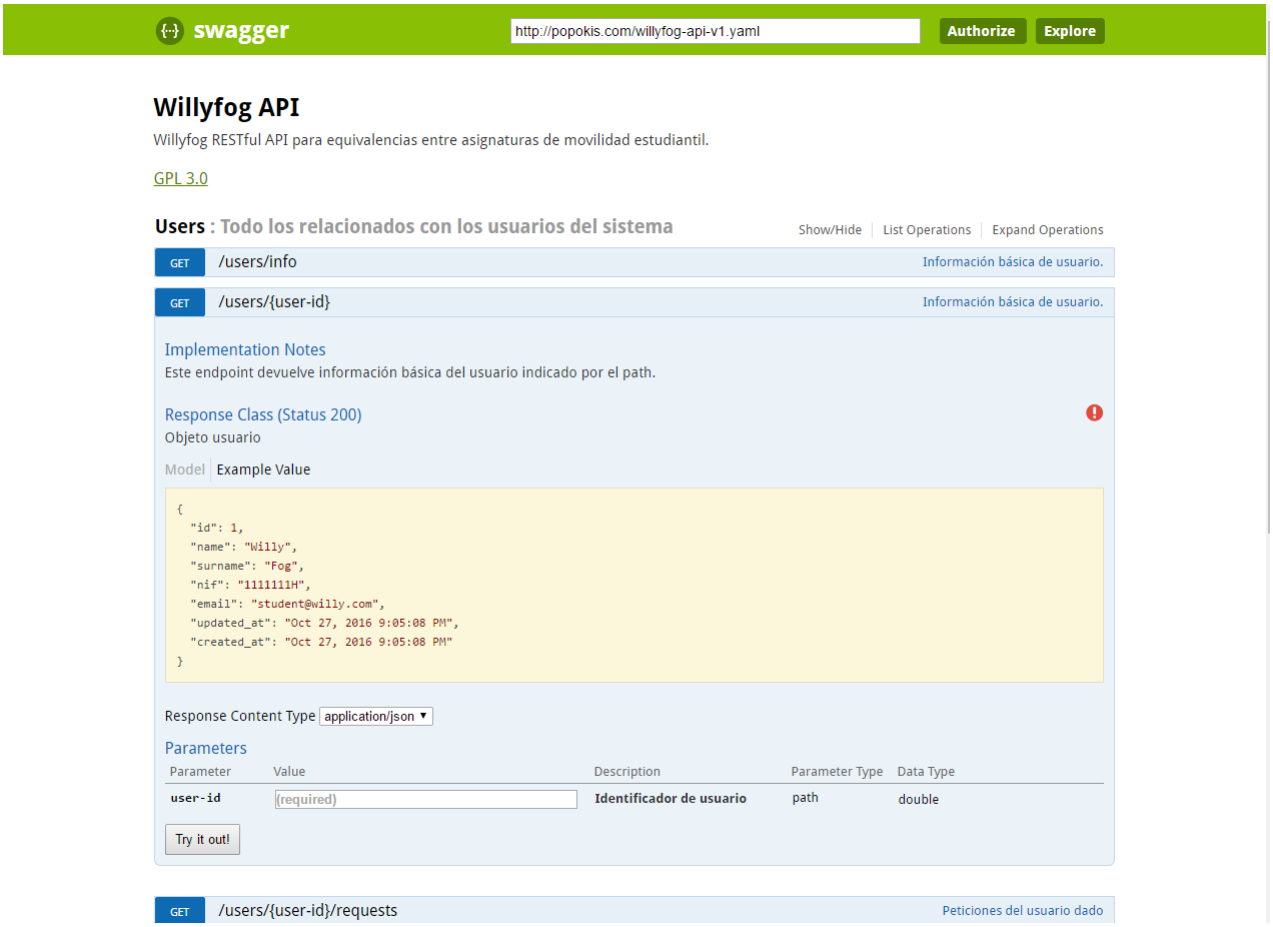


Figura 6: Apariencia de la documentación mediante *Swagger UI*.

4.4. Entorno de desarrollo

Este es un punto interesante de este proyecto, puesto que tanto mi compañero Nicolas Vargas Ortega (con quien he realizado este proyecto de forma conjunta) como yo, tenemos un punto de vista muy marcado al respecto.

Durante nuestro paso por la carrera descubrimos en profundidad las tecnologías de virtualización, y descubrimos el potencial que tenían a la hora de establecer entornos de prueba o *sandbox* que estaban totalmente aislados mediante el hipervisor de la maquina host que los contenía.

Más tarde descubrimos una utilidad, cada vez más famosa en los entornos empresariales como es **Vagrant** [25]. Esta herramienta permite la gestión de máquinas virtuales de forma que partiendo de una imagen base, o cajas (como se denominan en el argot de la misma), que no son más que una imagen de un sistema operativo como podría ser un Linux Server, y mediante el uso de *recetas*, que no son más que scripts (los cuáles pueden escribirse en multitud de lenguajes) que permiten definir el estado deseado de la máquina, en un par de sencillos comandos puedes tener un entorno de desarrollo perfectamente definido.

Otro añadido de esta herramienta, es que esta disponible en todas las plataformas, y mayoritariamente gracias a que la virtualización se puede realizar dede casi cualquier plataforma, se puede crear un entorno de desarrollo perfectamente definido el cuál es totalmente transportable entre sistemas operativos como Linux, Windows u OS X.

Pero una vez que empezamos a desplegar sistemas, y a bregar con sistemas que ya estaban en producción y que no tenían un entorno de pruebas o de desarrollo bien definido, descubrimos rápidamente la imperiosa necesidad de estas prácticas cuando se empieza un proyecto. De hecho, decidimos ir un paso más allá, y ya que las tecnologías actuales lo permiten fácilmente, establecimos que nuestros entornos de desarrollo tienen que ser casi equivalentes a quellos que encontramos en producción.

Esta afirmación se basa en varios puntos. En primer lugar gracias a la virtualización, no es difícil conseguir que nuestro sistema operativo (siempre Linux) en desarrollo sea el mismo que el que encontraremos en producción. Además, y esto se vuelve bastante necesario en lenguajes como PHP, el sistema depende en gran medida de los paquetes y modulos que hayan instalado en el mismo, es por eso que los scripts que provisionan nuestra máquina virtual se escriben en BASH y de forma independiente a la máquina virtual, para que de esa forma puedan ser usados tanto en desarollo como en producción.

Esto nos permite que ambos entornos (desarrollo y producción) sean imágenes casi especulares, y que cuando se sufre algún cambio, ya sea la instalación de un nuevo paquete en el sistema, o una nueva utilidad, esta misma se tiene que trasladar al etorno gemelo, para poder estudiar y convivir con las consecuencias que su añadido puede conllevar.

Por último, otra utilidad que nos facilita de sobremanera el desarrollo y testeo de

aplicaciones es **Docker** [26]. Con esta herramienta, podemos establecer procesos que son facilmente manejables y que no tienen persistencia de datos (a menos que se indique lo contrario). No llega al nivel de aislamiento de la virtualización, porque realmente son procesos que se ejecutan en la máquina "host"(aunque realmente no es buen término, ya que no hay *guest*) pero nos permite fácilmente, por ejemplo, crear instancias de base de datos desechables que son totalmente reseteables y que nos permiten tener datos de prueba.

Por supuesto, este es el uso que le hemos dados, pero hay casos probados de empresas que usan estos contenedores en producción y que lo usan de manera semejante a la que usamos Vagrant, para tener imágenes concretas de proceso y poder crearlos y destruirlos con mayor simplicidad, y a un menor coste computacional (puesto que no hay hipervisor de por medio), en nuestro caso, que no tenemos un sistema tolerante a fallos, ni de alta disponibilidad, se escapa a nuestro fin.

5. Conclusiones

Como primer punto a destacar, el desarrollo de este proyecto me ha servido para conocer más en profundidad procesos burocráticos, los problemas que entrañan y como poder resolverlos, principalmente, en el ámbito académico. Este era un dominio que nunca había tratado con anterioridad y que ha servido de escenario para poner en práctica los conocimientos adquiridos.

En segundo lugar, el desarrollo de una arquitectura totalmente distinta a lo que había aprendido en mis estudios me ha servido para conocer, poner en práctica, y desarrollar mis conocimientos sobre multitud de nuevas tecnologías. Abrir manuales, documentación, comunicarme con otros ingenieros e incluso organizarme con ellos ha sido una invariante en todo el proyecto y me ha hecho crecer como profesional, acercándome un poco más al mundo real de los proyectos *software* llegando a poner en práctica metodologías que utilizaré en mi carrera profesional.

En cuanto al sistema que hemos desarrollado, ha cubierto unas expectativas y requisitos iniciales, pero que en como cualquier proyecto *software*, irán cambiando con el paso del tiempo. Uno de los puntos más fuertes que en mi opinión tiene el proyecto es la de ser de código libre, por lo que en cualquier momento, gracias a su documentación y el uso de tecnologías más o menos extendidas, permite que cualquier desarrollador pueda retomarlo y adaptarlo a nuevas necesidades.

Tecnológicamente hablando, en un futuro cercano, una buena práctica sería separar la lectura de la base de datos en una API y la modificación de datos (creación, actualización y borrado) en otra distinta. Esto haría que la carga de trabajo más común (la lectura de la base de datos) quede totalmente aislada, y la tarea más crítica, de persistencia, quede concentrada en una sola parte.

Otra apartado de la arquitectura que podría ser mejorado es el de separar la tarea de autorización (controlar si la petición contiene un *access_token* válido en la cabecera) en un *proxy* externo a la API, de forma que a ésta solo lleguen peticiones autorizadas y así eliminar carga computacional innecesaria. Con esto conseguimos que el servidor de la API se encargue exclusivamente de la lógica de negocio.

También sería interesante subdividir las distintas funcionalidades de la aplicación en pequeños micro servicios para tener una arquitectura mucho más modular y tolerante a fallos. De esta forma, un proyecto que inicialmente estaba modelado para ajustarse a los requisitos de una sola facultad, podría llegar a satisfacer las necesidades de multitud de universidades sin importar la cantidad de estudiantes que hagan uso de ella.

Referencias

[1] *Planes de movilidad internacional. Universidad de Málaga.* http://www.uma.es/relaciones-internacionales/cms/menu/movilidad-estudiantes

[2] *Planes de movilidad nacional. Universidad de Málaga.* http://www.uma.es/sicue

[3] *¿Qué son los servicios RESTful? The Java EE 6 tutorial (Inglés).* http://docs.oracle.com/javaee/6/tutorial/doc/gijqy.html

[4] *¿Qué es OpenID?. OpenID (Inglés).* http://openid.net/get-an-openid/what-is-openid

[5] *¿Qué es SSO (Single Sign On)?. OAuth0 (Inglés).* https://auth0.com/docs/sso

[6] *Gravatar.* http://es.gravatar.com

[7] *Play Framework.* https://www.playframework.com

[8] *Lightbend.* http://www.lightbend.com

[9] *google/guice. GitHub.* https://github.com/google/guice

[10] *google/gson. GitHub.* https://github.com/google/gson

[11] *aaberg/sql2o. GitHub.* https://github.com/aaberg/sql2o

[12] *typesafehub/config. GitHub.* https://github.com/typesafehub/config

[13] *Slim Framework.* http://www.slimframework.com

[14] *guzzle/guzzle. GitHub.* https://github.com/guzzle/guzzle

[15] *Twig. SensioLabs.* http://twig.sensiolabs.org

[16] *Bootstrap.* http://getbootstrap.com/

[17] *jQuery.* https://jquery.com/

[18] *bshaffer/oauth2-server-php. GitHub.* https://github.com/bshaffer/oauth2-server-php

[19] *WSDL. W3.org.* https://www.w3.org/TR/wsdl

[20] *wsimport. Oracle Docs.* http://docs.oracle.com/javase/6/docs/technotes/tools/share/wsimport.html

[21] *RAML.* http://raml.org

[22] *Swagger.* http://swagger.io

[23] *Open API Initiative.* https://www.openapis.org

[24] *swagger-api/swagger-ui. Github.* https://github.com/swagger-api/swagger-ui

[25] *Vagrant.* https://www.vagrantup.com

[26] *Docker.* https://www.docker.com

[27] *sbt/sbt-assembly. GitHub.* https://github.com/sbt/sbt-assembly

[28] *swagger-api/swagger-codegen. Github.* https://github.com/swagger-api/swagger-codegen

[29] *Willyfog API documentación estática. GitHub Pages.* http://popokis.github.io/willyfog-api

Anexo

A. Manual de despliegue

La jerarquía de repositorios es la siguiente: hay un primer repositorio llamado *willyfog* el cual contiene documentación, *scripts* y en definitiva los archivos necesarios para etsablecer el entorno de desarrollo. Dentro, podemos encontrar la carpeta *projects* donde tendremos que situar el resto de repositorios para que el despliegue sea exitoso.

Una vez comentado esto, pasamos a detallar las instrucciones:

I. Entorno Vagrant

1. *git clone* el repositorio principal:

```
$ git clone https://github.com/popokis/willyfog.git ~/willyfog
$ cd ~/willyfog
```

2. Posteriormente, *git clone* el resto de repositorios que conforman el proyecto:

```
$ git clone https://github.com/popokis/willyfog-api.git
    projects/willyfog-api
$ git clone https://github.com/popokis/willyfog-openid.git
    projects/willyfog-openid
$ git clone https://github.com/popokis/willyfog-web.git
    projects/willyfog-web
```

3. Por ultimo, arrancar la máquina *Vagrant* y conectarnos a ella por *ssh*:

```
$ vagrant up
  [...]
$ vagrant ssh
  [...]
```

Una vez establecida la maquina de desarrollo, vamos a proceder con el despliegue de cada una de las partes del sistema.

II. willyfog-api

Conectando a la base de datos

Para conectar a la base de datos puede que te encuentres con dos opciones:

- Si despliegas la API con un *jar* (usando alguna herramienta como *sbt-assembly* [27] y ejecutándolo con *java* **dentro del entorno *Vagrant***. En este caso tendrás el servidor *MySQL* de manera local, así que no tendrás problema en conectarte.

- El caso en el que ejecutes el *jar* desde **fuera del entorno *Vagrant***, o más probablemente, porque estés desarrollando con un IDE como *IntelliJ* el cual se encarga de desplegar el archivo *jar* y lanzarlo por si mismo. En este caso tendrás problemas para alcanzar el servidor *MySQL* ya que **no lo tendrás de manera local**.

Creando un tunel SSH

No hay problema, vamos a crear un tunel *ssh* de manera que puedas conectarte a la base de datos desde fuera del entorno *Vagrant*. Ejecute este comando **desde la máquina host**:

```
$ ssh -f vagrant@192.168.33.10 -L 3307:localhost:3306 -N
```

Así, a partir de este momento tendrás al servidor *MySQL* escuchando localmente en el puerto 3307. Si quieres conectarte desde el *cli* de *MySQL*, tan solo tendrás que ejecutar **desde la máquina host**:

```
$ mysql -h 127.0.0.1 -P 3307 -uroot -proot
```

Usando *sbt dist*

Ya que es realmente simple, usamos *sbt dist* para construir nuestro ejecutable:

```
$ cd ~/willyfog/projects/willyfog-api
$ sbt
  [...]
  > dist
  [...]
  [success] Total time: XXXX s, completed Xxx XX, XXXX X:XX:XX PM
  > exit
```

Y una vez que ha terminado, tendrás un archivo *zip* que contiene los ejecutables en la ruta *target/universal/willyfog-api-1.0.zip*:

```
$ cd target/universal
$ unzip willyfog-api-1.0.zip
$ cd willyfog-api-1.0
```

Pudiendo ejecutarlo con tu *application secret* preferido:

```
$ bin/willyfog-api -Dplay.crypto.secret=abcdefghijk -Dhttp.port=7000 &
```

III. willyfog-openid

1. Instalar las dependencias.

```
$ cd ~/willyfog/projects/willyfog-openid
$ composer install
```

2. Establecer el archivo *constants.php*. Si no estas en producción, simplemente puedes renonmbrar el fichero *constants.php.example* a *constants.php*:

```
$ cp app/constants.php.example app/constants.php
```

3. Generar las claves públicas y privadas:

```
$ openssl genrsa -out data/privkey.pem 4096
$ openssl rsa -in data/privkey.pem -pubout -out
    data/pubkey.pem
```

IV. willyfog-web

1. Instalar las dependencias.

```
$ cd ~/willyfog/projects/willyfog-web
$ composer install
```

2. Establecer el archivo *constants.php*. Si no estas en producción, simplemente puedes renonmbrar el fichero *constants.php.example* a *constants.php*:

```
$ cp app/constants.php.example app/constants.php
```

3. Enlazar la clave pública del servidor OpenID (para poder manejar los JWT):

```
$ cd data
$ ln -s ../../willyfog-openid/data/pubkey.pem
```

V. Dominios

Recuerda añadir los distintos dominios al archivo */etc/hosts*:

```
$ echo "192.168.33.10  willyfog.com api.willyfog.com
    openid.willyfog.com" | sudo tee -a /etc/hosts
```

B. Manual de la API

Debido a la extensión de la documentación, hemos decidido no incluirla en este documento. Para poder visualizarla hay varias opciones:

- Versión estática: la versión HTML de la documentación swagger, generada con la herramienta *swagger-codegen* [28] esta disponible en la página del proyecto en *github pages* [29].
- Versión dinámica: para poder visualizarla es necesario recurrir a la herramienta *Swagger UI*. En el repositorio de *GitHub* [24] se puede obtener una versión *standalone* (preparada para ser visualizada por un navegador) y la API tiene un *endpoint* */api/v1/docs* a través del cuál se puede obtener el fichero YAML de especificación.

C. Manual de usuario de la aplicación web

El primer paso para poder usar la aplicación web es la de iniciar en el sesión. En este momento se presentan dos caminos: estar visitando por primera vez la aplicación o ingresar siendo un usuario registrado en el sistema.

I. Usuario de nuevo ingreso

Para poder acceder al **registro de estudiante** (el único que tiene acceso público) hay que dirigirse a la sección "Register" en la cabecera, parte superior derecha de la página web.

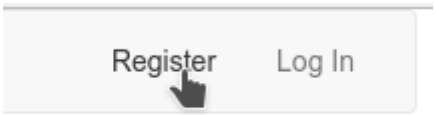


Figura 7: Sección Register en la cabecera.

Entonces accederemos al formulario de registro para estudiantes. Tendrás que completarlo con los datos correspondientes, como por ejemplo, el grado al que perteneces como estudiante en tu Universidad. Para completar el proceso pulsa en "Submit".

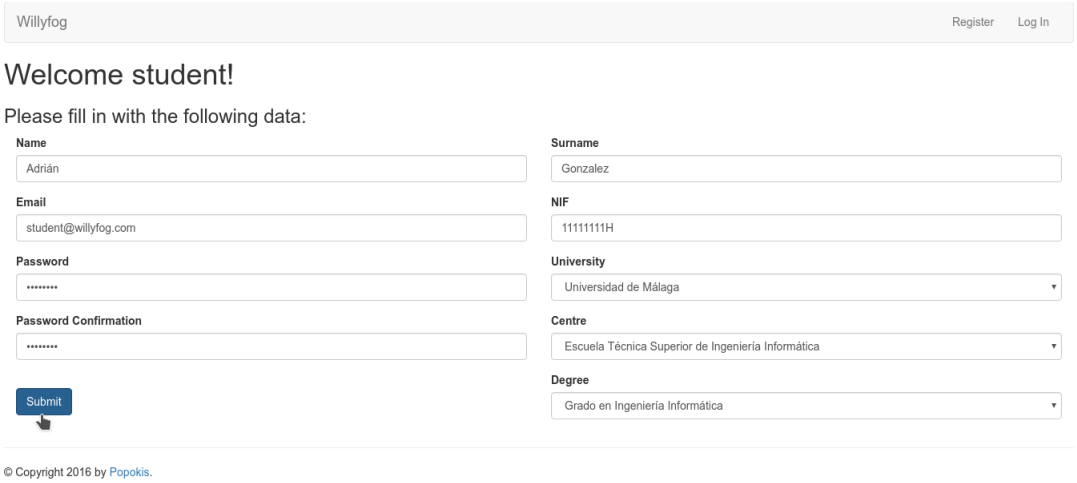


Figura 8: Formulario de registro.

En el caso de que los datos introducidos sean correctos, serás redirigido a la página principal con un mensaje indicándote que ya formas parte del sistema. En el caso contrario, volverás al formulario con un mensaje indicando el error al procesar el formulario para que puedas corregirlo.

A partir de este momento, podrás seguir el segundo camino que se puede dar en el sistema, que es el de un usuario ya registrado.

II. Inicio de sesión

En el caso de que seas un usuario ya registrado en el sistema (sea cual sea el rol que tengas en la aplicación) podrás acceder al mismo pulsando en la sección de "Log In" en la cabecera, en la parte superior derecha.

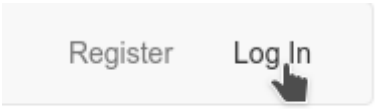


Figura 9: Sección Log In en la cabecera.

Serás redirigido al servidor OpenID para el inicio de sesión, donde tras introducir tu correo electrónico y contraseña (combinación con la cual te registraste en el sistema) podrás acceder inmediatamente a la aplicación con los privilegios correspondientes.

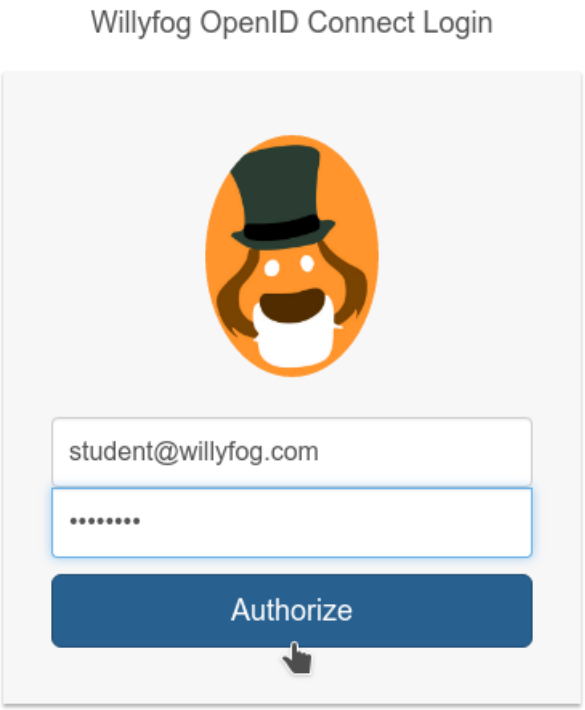


Figura 10: Inicio de sesión OpenID.

Una vez iniciada a la sesión, llegará a la página principal de la aplicación, donde verá sus peticiones abiertas en el sistema dependiendo del rol que tenga dentro de la aplicación. Se mostrará una tabla donde las peticiones se dividen en aquellas que estan pendientes de moderar y aquellas que ya han sido tratadas por los reconocedores.

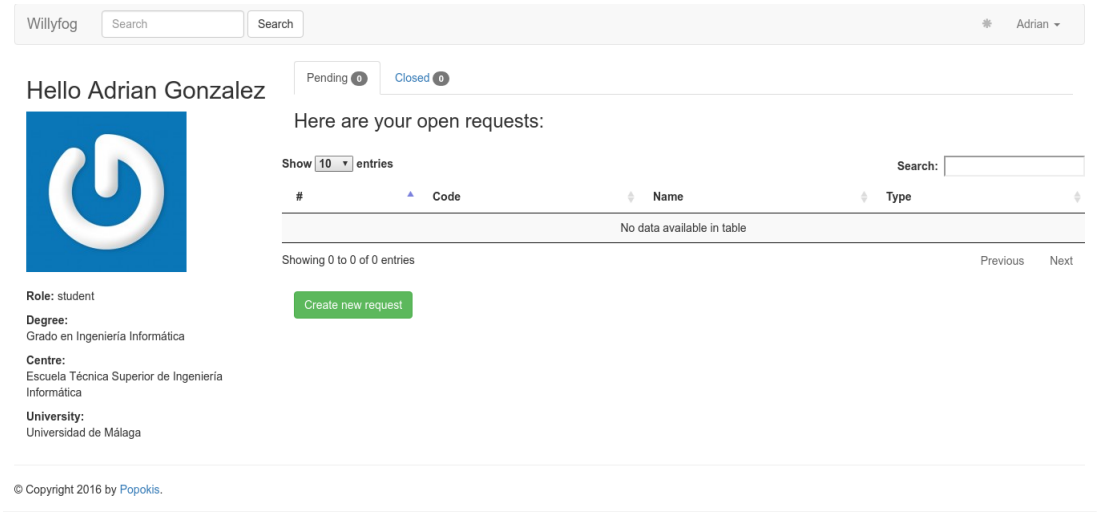


Figura 11: Pantalla principal una vez iniciada la sesión.

En parte superior puede observar el buscador de asignaturas, en el que introduciendo un parámetro de búsqueda, podrá realizar una búsqueda entre la tabla de equivalencias entre asignaturas actual.

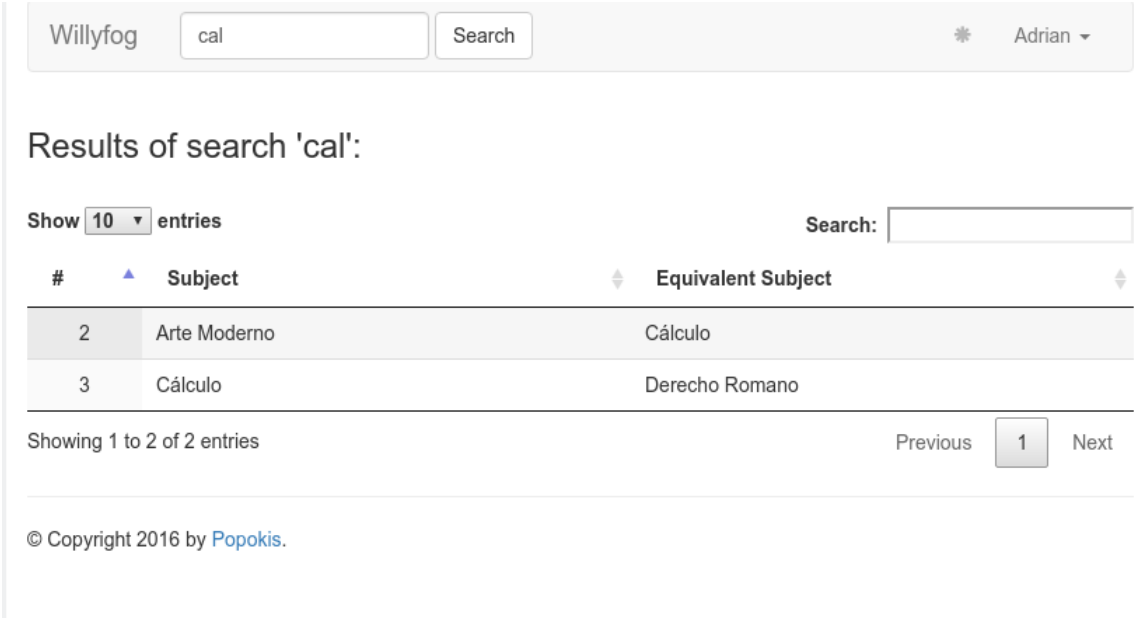


Figura 12: Buscador sobre la tabla de equivalencias.

En la parte inferior de la página principal, y en el caso de tener el rol de estudiante, podrá acceder a crear peticiones. Accederá a un formulario que tendrá que rellenar con la información necesaria, con datos como la asignatura de origen, las de destino, tipo de plan de movilidad, etc. Un detalle destacable es el de la posibilidad de añadir más de una asignatura de destino con los votones de "Add" y "Delete" en la parte superior derecha de la sección de "Destination".

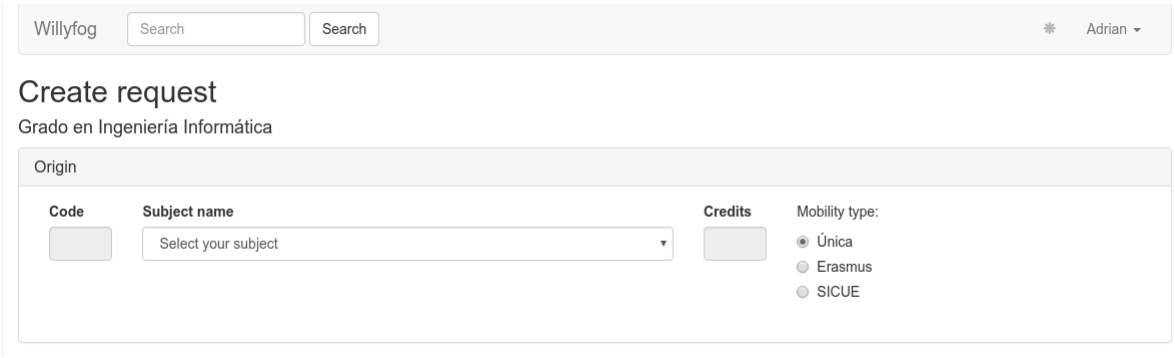


Figura 13: Sección de origen en la creación de una petición.

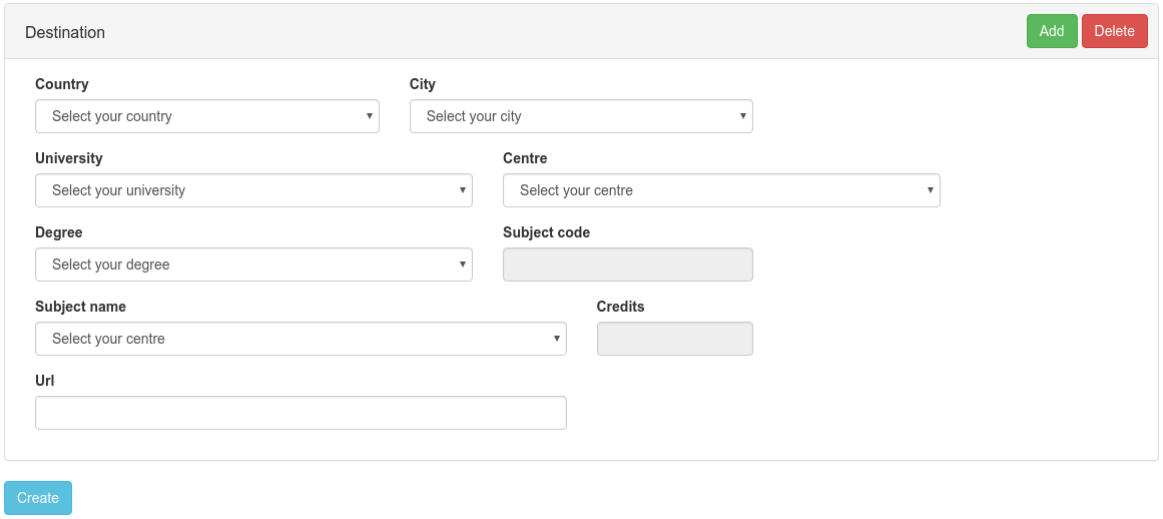


Figura 14: Sección de destino en la creación de una petición.

Una vez creada una petición, aparecerá en la página principal, y podrá ser consultada en cualquier momento para ver el estado de la misma (el cual aparecerá en la parte superior derecha). Las peticiones también pueden ser comentadas por usuarios del sistema en la parte inferior.

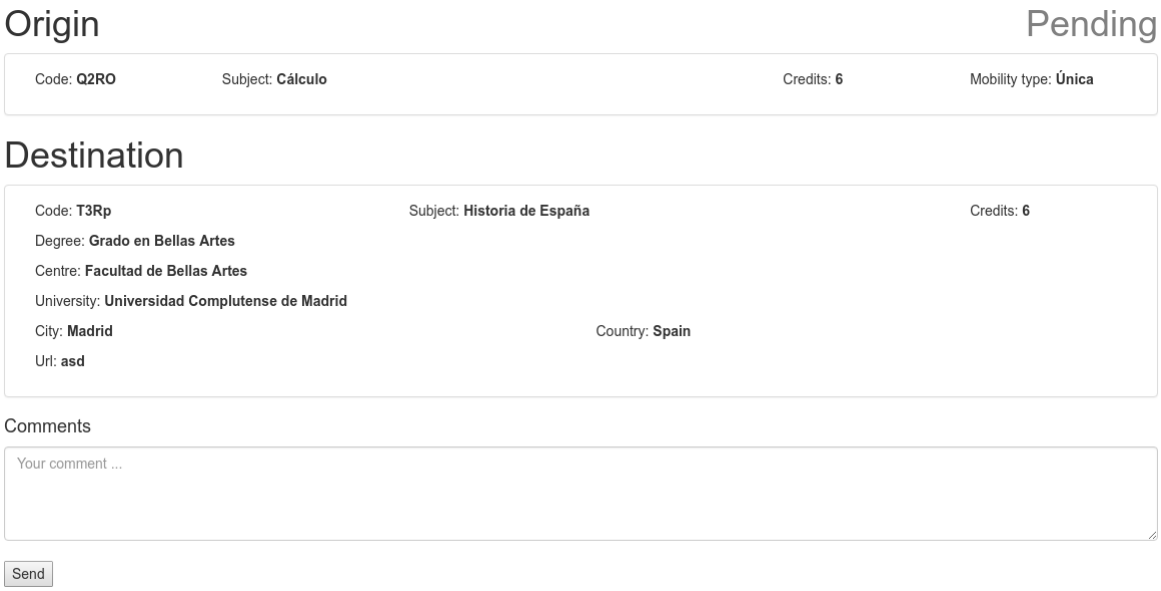


Figura 15: Información de una petición.

Cualquier usuario en el sistema podrá acceder a sus notificaciones en la parte superior derecha, pulsando en el icono del asterisco. El sistema irá generando notificaciones para los distintos usuarios dependiendo de las acciones que estos tomen.

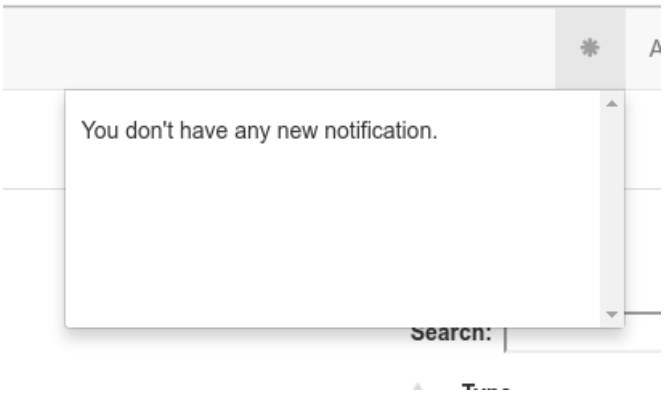


Figura 16: Notificaciones de un usuario.

III. Profesores de reconocimiento

En esta sección vamos a proceder a comentar las particularidades de los usuarios cuyo rol es el de profesor de reconocimiento. Como hemos comentado, el inicio de sesión es unificado por lo que su camino hasta entrar en la aplicación no será distinto al de un estudiante.

La primera particularidad que observarán es que los profesores de reconocimiento no pueden crear peticiones. En cambio, ellos si podrán moderar las peticiones en el sistema. Es por ello que las peticiones que verán en su índice en la pantalla principal, serán aquellas peticiones que tienen como objeto las asignaturas que ellos reconocen.

Al acceder a la información de las mismas, en la esquina superior derecha, en el caso de que una petición este pendiente de reconocimiento, encontrarán dos botones para poder aceptar o rechazar peticiones. También podrán ver qué usuario creó esa petición.

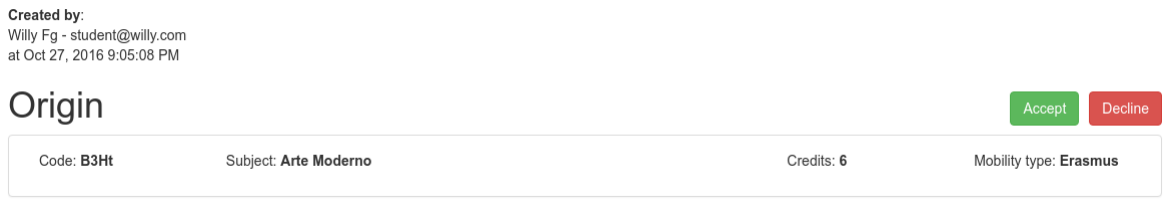


Figura 17: Botones para poder moderar una petición.

Otra particularidad es que tendrán una sección en la cabecera (justo al lado del buscador) para poder ver qué asignaturas son las que reconoce.

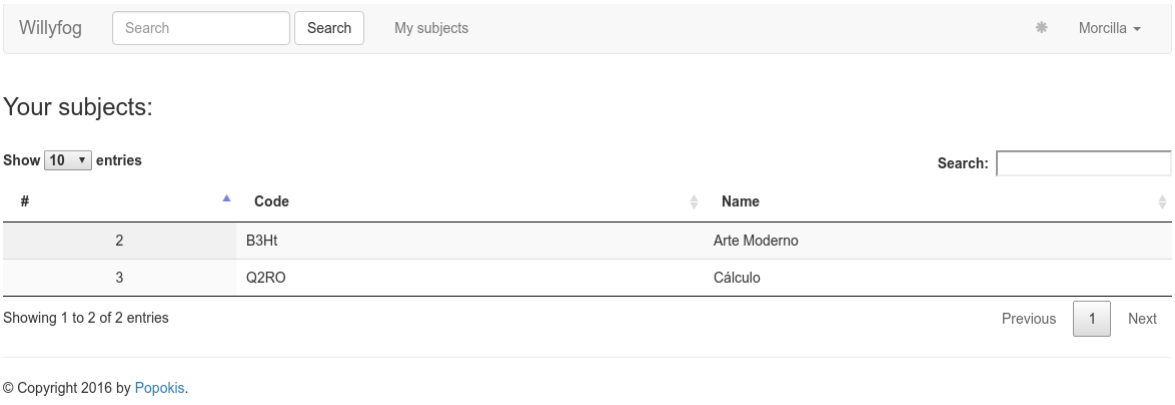


Figura 18: Índice de asignaturas que reconoce el usuario.

IV. Coordinadores de centro

En el índice de peticiones, un usuario cuyo rol es el de coordinador de centró podrá ver todas las peticiones cuyas asignaturas objetos pertenecen a un grado de su centro.

A diferencia de un profesor reconocedor, un coordinador no podrá modificar el estado de una petición, pero también podrá ver el usuario el cual creó la petición.

Un coordinador de centro tendrá dos secciones extra (en la parte superior, junto al formulario de búsqueda) que no tienen el resto de roles de la aplicaciones. Una de ellas es el registro de profesores de reconocedores. En ella podrán dar de alta usuarios cuyo rol será el de profesor de reconocimiento.

La otra sección es un índice de los profesores de reconocimiento que se encuentran en el sistema. En tal índice los coordinadores podrán asignar y eliminar asignaturas que reconocen dichos profesores.

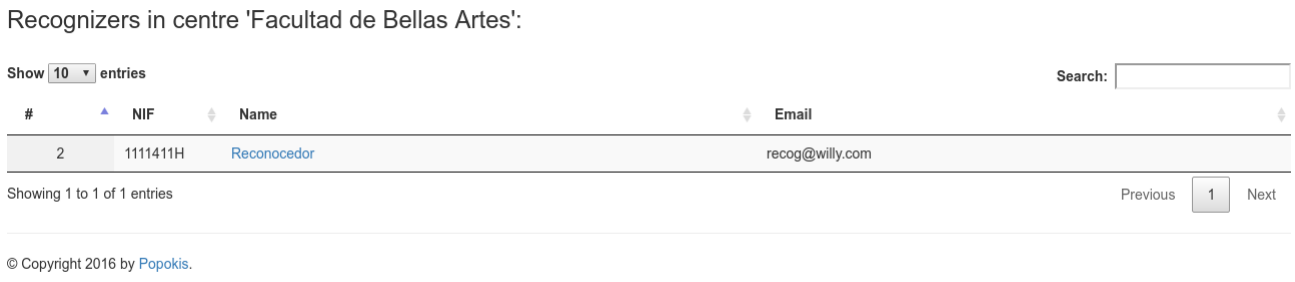


Figura 19: Índice de reconocedores.

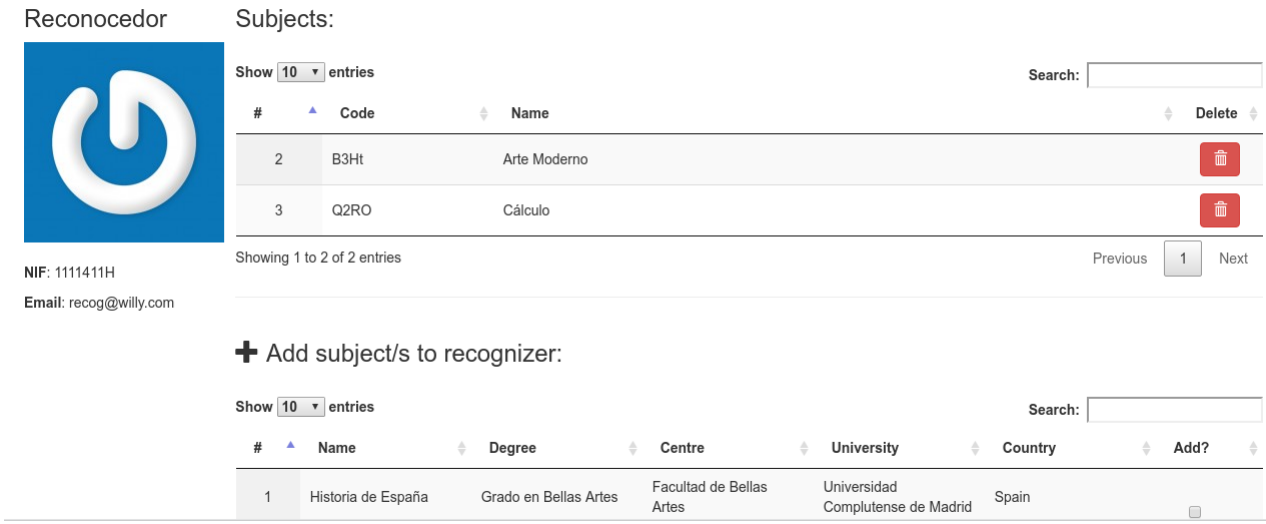


Figura 20: Añadir/eliminar asignaturas a un reconocedor.